

# Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

284

---

A. Kündig R.E. Bühler J. Dähler (Eds.)

## Embedded Systems

New Approaches to Their Formal Description and Design  
An Advanced Course  
Zürich, Switzerland, March 5–7, 1986

---



Springer-Verlag

Berlin Heidelberg New York London Paris Tokyo

**Editorial Board**

D. Barstow W. Brauer P. Brinch Hansen D. Gries D. Luckham  
C. Moler A. Pnueli G. Seegmüller J. Stoer N. Wirth

**Editors**

A. Kündig  
R. E. Bühner  
J. Dähler  
Institut für Elektronik  
ETH-Zürich, ETH-Zentrum  
CH-8092 Zürich, Switzerland

CR Subject Classification (1987):D.1–3, C.2.2, E.4, B.7.2

ISBN 3-540-18581-X Springer-Verlag Berlin Heidelberg New York

ISBN 0-387-18581-X Springer-Verlag New York Berlin Heidelberg

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in other ways, and storage in data banks. Duplication of this publication or parts thereof is only permitted under the provisions of the German Copyright Law of September 9, 1965, in its version of June 24, 1985, and a copyright fee must always be paid. Violations fall under the prosecution act of the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 1987

Printed in Germany

Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.  
2145/3140-543210

## Preface

This book contains the written form of some lectures given in Spring 1986 at ETH Zürich in the framework of an Advanced Course on **New Approaches to the Architecture and the Design of Embedded Systems**. It was felt at that time - and it is still felt - that there is a considerable gap between newer research results in the area of system architecture and its description on the one hand, and actual application of these results in the development of real time/process control systems on the other hand. When organizing the Advanced Course, it was clear from the outset that this gap cannot be bridged once and forever in a single event. Nevertheless, we hope that the presentations as well as the many contacts among lecturers and participants helped to stimulate thinking along some new lines.

Since the lectures of the architecture part of the course have been published in different form elsewhere, the present Lecture Notes concentrate on the design aspect of embedded systems. It is recognized that development of such systems is still dominated by progress made in the semiconductor field. Integrated circuits of ever increasing complexity allow for the design of even more complex systems, interacting in space and time with environments being themselves of high complexity. The contributions in this book tackle the question of how this complexity may be harnessed on rather different levels. Wolfgang Fichtner presents the state of the art in IC technology and the approaches taken in this field to master design problems at the hardware system design level. Wolfgang Reisig then presents a methodology based on Net Theory, a discipline which emerged in the 70's from Petri Nets. His model of concurrency may well be applied at many different levels, ranging from hardwired logic to the description of the top level functionality as contained in requirements specifications.

As it is increasingly felt that a real step forward in understanding, describing and designing complex systems must be based on sound theoretical ground, the four contributions by Raymond Boute and Simon Peyton-Jones form the core part of the text. It is hoped that their compilation of many interesting theoretical concepts may stimulate practitioners in casting these approaches into powerful new tools.

Probably the outstanding field where formal description methods have already found their way into widespread applications is the one related to protocol and interface design for communication systems (e.g. in the context of the ISO/OSI model). Harry Rudin gives an overview of the state in this field. The text is rounded up with a contribution by Jochen Ludewig which serves two purposes: It shows that beside formal specification methods, semiformal methods are of considerable importance as well. Also, an overview of some proven practical methods and the available tools is given.

Concluding this preface, it is a must to thank the numerous people who have contributed to both the organization of the Advanced Course and the compilation of these Lecture Notes. Above all, due credit must be given to the lecturers who were willing to take up the burden a second time, transforming their interesting oral presentations into a publishable format.

Albert Kündig, Richard E. Bühler, Jacques Dähler

## Contents

|                                                                 |     |
|-----------------------------------------------------------------|-----|
| A Note on the Meaning of "Embedded Systems"                     |     |
| A.T.Kündig, ETH Zürich / Switzerland . . . . .                  | 1   |
| Design of VLSI Systems                                          |     |
| W.Fichtner, ETH Zürich / Switzerland . . . . .                  | 6   |
| Embedded System Description Using Petri Nets                    |     |
| W.Reisig, GMD Sankt Augustin / Germany . . . . .                | 18  |
| Elements for the Formal Description of Systems                  |     |
| R.T.Boute, University of Nijmegen / The Netherlands . . . . .   | 63  |
| An Introduction to System Semantics                             |     |
| R.T.Boute, University of Nijmegen / The Netherlands . . . . .   | 91  |
| Functional Formalisms for System Description:                   |     |
| Current Research Topics                                         |     |
| R.T.Boute, University of Nijmegen / The Netherlands . . . . .   | 108 |
| Tools for Protocols Driven by Formal Specifications             |     |
| H.Rudin, IBM Research Laboratory Zürich / Switzerland . . . . . | 127 |
| Functional Programming Languages as a Software Engineering Tool |     |
| S.L.Peyton Jones, University College London / England . . . . . | 153 |
| Practical Methods and Tools for Specification                   |     |
| J.Ludewig, ETH Zürich / Switzerland . . . . .                   | 174 |

# A NOTE ON THE MEANING OF "EMBEDDED SYSTEMS"

Albert T. Kündig  
Institut für Elektronik / Institut für Informatik  
Federal Institute of Technology (ETH)  
Zürich, Switzerland

## INTRODUCTION

As it is with many often used terms in informatics and computer applications: The longer one is looking for a proper definition, the more one might get confused! This situation also arose when we discussed the meaning of subjects like "real time systems", "distributed systems" etc. in setting up new courses at ETH in these fields. Finally, we settled on the notion of "**Embedded Systems**", expressing thereby our aim to treat computer based systems embedded in real life processes. However, it turned out that the term was originally coined by people using it for certain military applications, eg. weapons control or, in a broader sense, military command/control/communications systems. Though such systems quite naturally fall into the class of systems we considered (e.g. regarding real time requirements, reliability, complexity etc), we certainly did not want to restrict our teaching to military systems. The aim of this note is therefore to attach a broader meaning to the term "**Embedded Systems**", by giving a more or less informal description of the class of systems treated in these Lecture Notes.

## SOME CHARACTERISTICS OF EMBEDDED SYSTEMS

We consider electronic systems embedded within a given plant or external process with the aim of influencing this process in a way that certain overall functional and performance requirements are met. The external process usually consists of many more or less independent sub-processes distributed in space such that communication between them takes up a non negligible time. Often, alternative names are given to such systems, such as process control systems, real time systems, distributed systems etc. The notion "embedded systems" (ES) is preferred and will now be described in a way that a broad class of systems is encompassed, ranging in principle from simple microprocessor based apparatus to complex systems controlling large plants, aircrafts and the like. The description will first characterize embedded systems in terms of their interactions with the outside process, and deduce thereof certain properties of the internal structure and operating principles, observing some assumptions about available technologies.

- (1) An ES is an electronic system embedded within a given plant or **external process**. The external process comprises both a physical system (usually consisting of different subsystems) and also humans performing some supervising or parameter setting tasks. Essential to the external physical system is that it usually obeys its own dynamics and does not allow restoration of a previous state in case of failures. Coupled with this property is the requirement that the ES has to respond to certain inputs within prescribed time constraints (**real time requirement**).

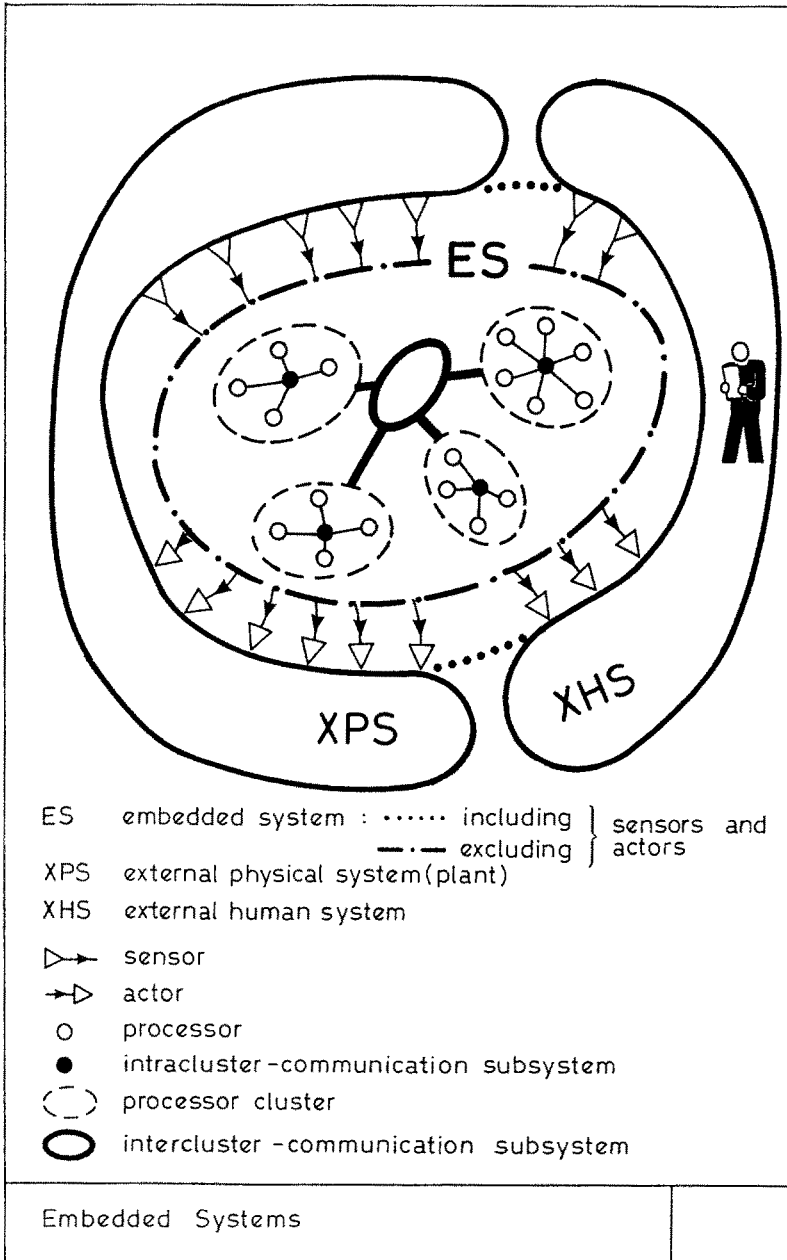


**Figure 1** A coarse reference model for Embedded Systems

- (2) The coupling between external processes and the ES is through suitable **sensors** and **actors**. There is actually a grey zone in terms of "preprocessing" or "postprocessing": E.g. significant external state changes may only be reported as an "event" to the ES, or inversely, external events may be summed (integrated) to yield quasi-continuous input variables.

Both **input and output data** may be of different type:

- they may correspond to a spontaneous event with predetermined meaning (e.g. "a knob is pushed")
- they may correspond to a spontaneous transmission of the value of some state variable
- they may represent continuous signals, usually suitably sampled and coded (e.g. PCM, DPCM, etc.)

Real time requirements prescribe certain time relations between particular input signals and particular output signals.

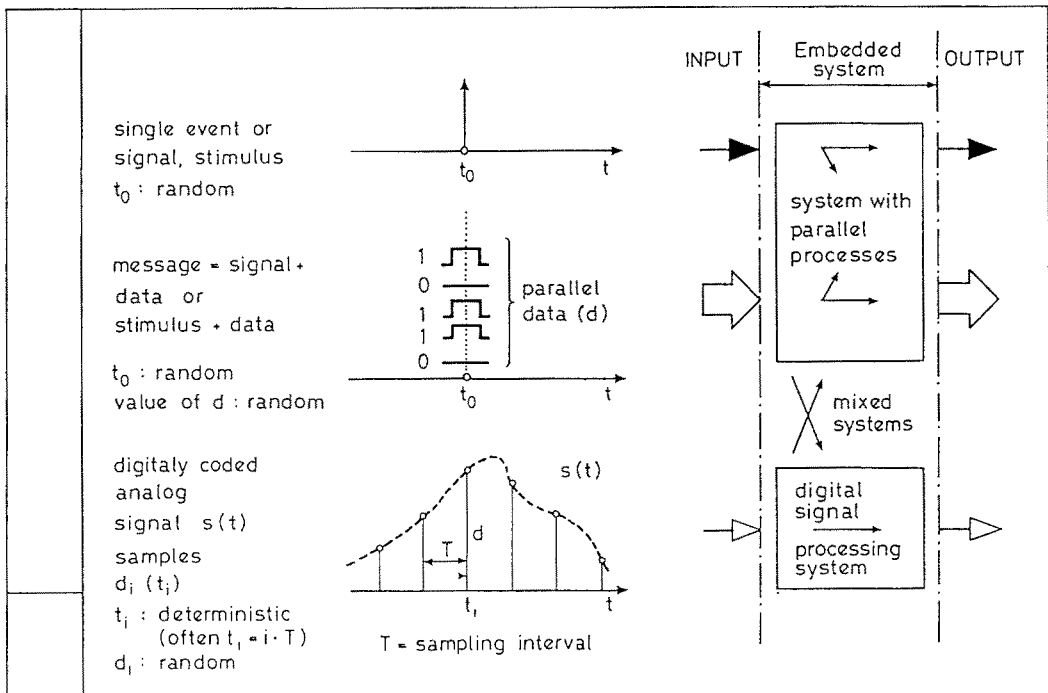


**Figure 2** Input/Output relations for Embedded Systems

(3) Sensors and actors may be **distributed** in a way that their connection to the ES requires more elaborate communication means than just a direct connection to a computer I/O subsystem or a computer bus. Also, the ES itself may assume the character of a **distributed system**:

- **Clusters of processors** may be separated from each other by distances which forbid interprocessor communication by shared memory; rather some **message passing** through a **common communication subsystem** is assumed
- within a cluster of processors, shared memory may be available. However, its implementation may encompass also solutions whereby part of each processors individual memory is "linked" by high performance **intracluster communication subsystem**
- It is assumed that the availability (reliability) of the intra-cluster communication subsystem is much higher than the one for intercluster communication; i.e. while high reliability of intracluster communication is achieved by hardware measures, some software (i.e. a protocol) may be needed (though preferably at a low level) to ensure adequate intercluster communication reliability.

(4) Most ES must fulfill stringent **reliability requirements**, hopefully detailed according to a **set of functions** to be performed. Different reliability requirements are usually valid for different functions, and often, some "fail soft" character of the ES may be tolerated (i.e. less critical functions may more often be subject to errors or may tolerate longer repair times).

In order to fulfill the reliability requirements, data and/or processes may be **replicated**, both within a cluster and also among clusters. Note that with regard to replication, there exists again a "grey zone" between the ES and the external plant in that sensors and/or actors may advantageously be replicated as well in certain cases.

(5) Both at the cluster level and also at the system level, the ES **may not be homogeneous**:

- There may be different, e.g. **specialized processors** within a cluster (typical examples: attached signal processors, communication preprocessors etc.)
- The overall set of functions performed by one cluster may be significantly different from other clusters (**specialized clusters**).

Also, groups of sensors and/or actors may be connected to only one or a few cluster(s). Note that the external process (or plant) is an abstraction in the sense that it can be decomposed into a plurality of more or less closely coupled individual processes. Note also that the physical external processes are not necessarily decoupled from the processes representing the human intervention since the human input signals may depend on a direct observation of the physical process, and in turn, the external process may impact the ability of the personnel to interact with the system (e.g. excessive heat may force the personnel to leave).

- (6) For a certain application, the meaning of the term **system** ought to be carefully defined, and in conjunction also terms like "modularity", "flexibility" and the like. In this context, an ES may or may not be required to possess one or more of the following characteristics:

- Depending on the application, different types of ES may be constructed out of a set of given **hardware and software modules**
- A specific **class of applications** may be covered by an abstract system, and a particular runtime system can then be regarded as a particular instance of the given system class.

Note that this discussion is leading to problems like versions or generations of modules and their compatibility, version control, configuration management etc. Quite generally, methods to master the development process from initial user requirement through to successful operation and maintenance (**life cycle**) are needed. Often, the **development system** and the **target system** may be distinctly different.

- (7) Some ES may require their **expansion both in processing power and even functionality at runtime** (as e.g. for telecommunication networks), may be at the cost of defined short interruptions of service (or a subset of the service). Typically new "clients" within the external plant will be installed at runtime.
- (8) In contrast to most standard DP systems or workstations, many ES are subject to extreme **environmental conditions** (heat, humidity, temperature changes etc.). Also, stringent requirements regarding weight, power consumption etc. may exist.
- (9) Some functions of an ES may call for **extremely high processing power and/or special processor architectures**, e.g. for artificial vision, voice processing, knowledge handling, etc. Also, providing **efficient interfaces to human operators** or users of the ES could require considerable resources and special preprocessing functions. Since it is difficult to predict human reactions, the aid of **fast prototyping** tools in the early development phases is certainly helpful.

# Design of VLSI Systems

*W. Fichtner*

Institute for Integrated Systems  
Swiss Federal Institute of Technology  
Gloriastr. 35  
CH 8092 Zurich  
Switzerland

## ABSTRACT

VLSI chips have become an important part in the design of many large electronic systems. The growing complexity of VLSI chips creates a need for improved design methodologies and more powerful CAD environments.

The material presented in this chapter provides an overview of current methodologies for the design of VLSI chips. Particular emphasis has been given to provide realistic and up-to-date examples of the various design methods.

## 1. Introduction

During the last three decades, we have witnessed an unprecedented growth in the field of digital technology. Digital integrated circuits have become a major part of all electronic systems ranging from household systems to communications equipment and computers. VLSI (Very Large Scale Integrated) chips have become standard subcomponents and may be considered to be prototypes of embedded systems.

The rapid growth of the microelectronics field was fueled by the advances in semiconductor technology in the same period. Modern technology has made it possible to place more than one million devices on a single piece of silicon (a chip). At this point in time, there is no major technological obstacle preventing a further complexity increase by several decades.

While the limits of technology seem far in the future, the design of today's highly complex chips generates difficult and sometimes unsurmountable problems. While highly complex systems such as bridges, skyscrapers or power stations have been successfully designed in the past, VLSI systems are sufficiently different in that their design involves multiple levels of hierarchy in different domains.

In [1] C. Sequín has made the following statement that characterizes the current situation extremely well:

"Nobody would dare to insert a million discrete devices into a large chassis using discrete point-to-point wiring. Large systems are broken down into sub-chassis, mother-boards, and module-boards carrying the actual components. This physical partitioning encourages careful consideration of the logical partitioning and the interfaces between the modules at all levels of the hierarchy. Since such systems are typically designed by large teams, early top-down decisions concerning the partitioning and the interfaces must be made and enforced rigidly - for better and for worse. This keeps the total complexity in the scope of each individual designer limited in magnitude, and thus manageable.

VLSI permits the whole system to be concentrated in a basically unstructured domain on a single silicon chip which does not a priori force any partitioning or compartmentalization. On the positive side, this freedom may be exploited for significant performance advantages. On the negative side, it may result in a dangerous situation where the complexity within a large unstructured domain simply overwhelms the designer".

The unstructured nature of a VLSI system has lead to the development of several design methodologies. This chapter contains an overview of the design techniques in use today.

In the next section, a short summary of the historical development of device technology and design efforts should illustrate the enormous achievements made during the decade of the seventies. Section 3 gives an indication of the hierarchy levels in VLSI design.

Section 4 describes the various design methodologies developed that allow the design to - more or less - cope with the VLSI complexity issue. The last section provides a summary and an outlook into the future.

## 2. Historical Development

An integrated circuit consists of an assembly of tightly packed, interconnected semiconductor devices (transistors, diodes, capacitors) on a common carrier substrate. The successful development of the integrated circuit area has been largely due to technological advances that allow a continuing decrease in feature sizes. From the early days of IC technology, the MOSFET (Metal-Oxide-Semiconductor Field-Effect Transistor) has been the basic building block of an integrated digital circuit. While the concept of the field effect transistor was already conceived in the 1930s, it was not until 1959 that the first experimental device has been realized by Kahng and Atalla [2]. The conceptual simplicity of this device together with its ease of integration have made it to *the* VLSI element of the eighties. Figure 1 shows a photograph of the first MOSFET realized.

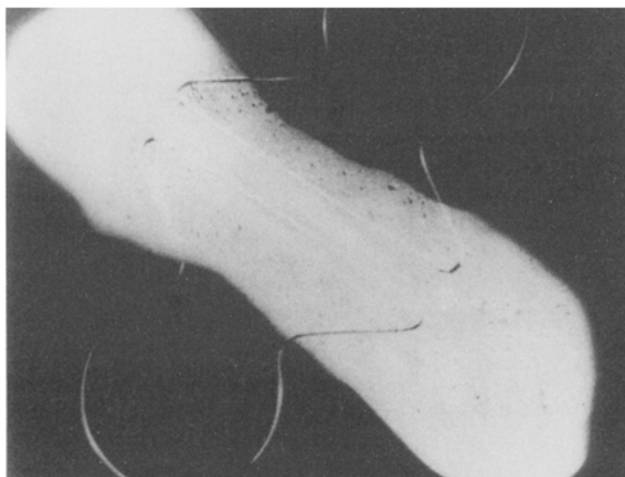


Fig. 1 First MOSFET fabricated using a thermally oxidized silicon substrate. [2].

The two distinct areas at the top and the bottom are the source and the drain of this transistor, and the light gray area constitutes the gate. The device had a channel length greater than  $20\mu\text{m}$  and a gate oxide exceeding  $1000\text{\AA}$ . Although present day MOSFETS are considerably smaller, the choice of silicon and the use of a thermally grown  $\text{SiO}_2$  layer remain the preferred choices.

Figure 2 shows a photograph of a test chip containing MOSFETs with active feature sizes of  $0.5\mu\text{m}$ .

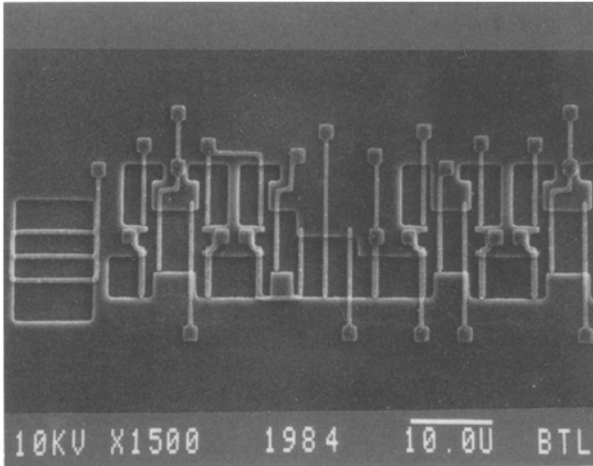


Fig. 2 NMOS circuit with  $0.5\mu\text{m}$  feature sizes. [3]

During the sixties, "integrated" circuit design consisted of the creation of geometric patterns in sheets of mylar. These patterns were photographically reduced to provide simple masks for IC fabrication.

Parallel to the advances in IC technology, enormous progress was made in IC design since the early seventies. The reason for this progress was mainly due to the advent of more user-friendly computers, combined with the development of the first CAD techniques such as pattern digitization.

Figure 3 shows a chip photograph of Intel's 4004, the world's first microprocessor introduced in 1971. Designed in  $12\mu\text{m}$  rules, the chip integrated 2300 transistors. It contained a four-bit parallel adder, 16 four-bit registers, an accumulator, and a LIFO stack. It was integrated in silicon gate MOS technology. The design time of this chip was 9 months for one designer. One notices the "random" wiring of this chip.

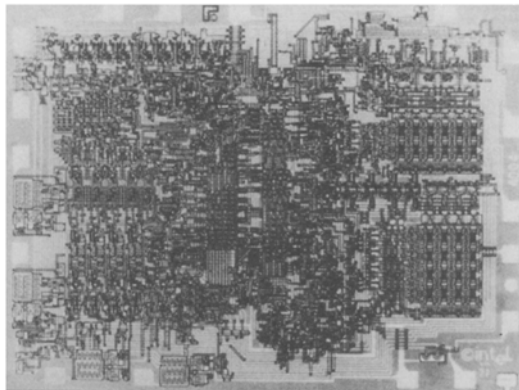


Fig. 3 Chip photograph of Intel's 4004 microprocessor [4]

By 1978, Intel introduced its successful 16-bit MOS processor, called the 8086 (see Fig. 4). It was designed by five engineers. The chip has a complexity of 29,000 transistors, over 12 times as many as the 4004.

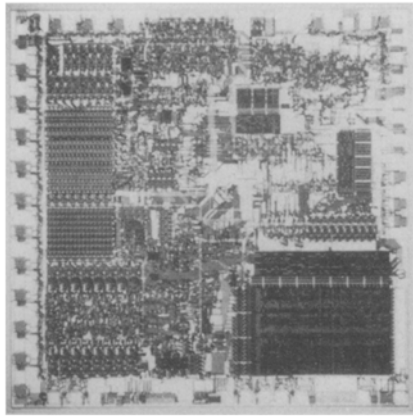


Fig. 4 Chip photograph of Intel's 8086 microprocessor [4]

Only a few years later, the complexity of chip design had increased another decade. By 1981, HP introduced its FOCUS CPU chip (Fig. 5) [5]. The FOCUS chip is 32-bit CPU containing about 450,000 transistors fabricated in a  $1\mu\text{m}$  silicon gate process with three layers of interconnect.

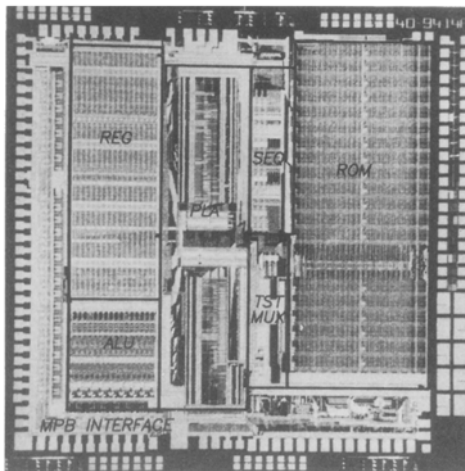


Fig. 5 Chip photograph of HP's FOCUS CPU chip [5]

Because the design of the CPU chip was undertaken with a relatively small group and in parallel with an emerging technology, a structured design approach was employed. A substantial effort was invested in reducing the overall design task by reusing as many cells as possible. The entire CPU chip was fabricated with 100 unique cells whose performance was simulated on desktop computer systems. The computer models were simplified and could simulate a maximum of 25 transistors. The complete set of CAD tools ran on four machines. Only one of these machines was a mainframe used for the final design-rule checking of the chip.

The above examples were chosen to illustrate the enormous increase in complexity that was achieved in the past 15 years. In the following, an overview of the various VLSI design styles in use today.

### 3. The VLSI Design Process

Figure 6 should give a rough indication of these levels. Three different domains - function, structure and the physical domain - can be distinguished in this figure.

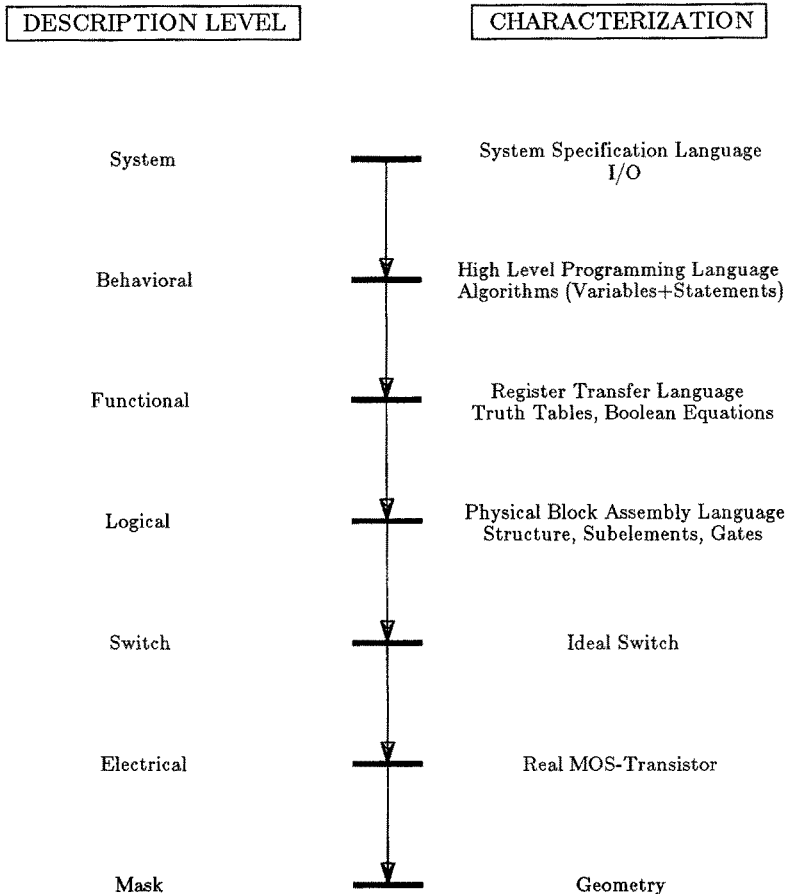


Fig. 6 The VLSI Design Hierarchy

The description provided by Fig. 6 views VLSI design as a series of design steps leading from the system level via the functional and architecture level to the logic and the circuit design.

### 4. VLSI Design Styles

In the period of the seventies, several design styles have been developed to utilize the possibilities offered by the new technologies. Each of these choices offers the design trade-offs between chip density and design time. We shall present typical examples of these option in the following subchapters.

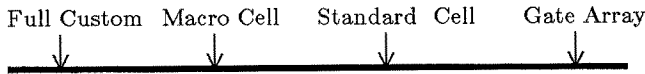


Fig. 7 The VLSI Design Spectrum

Figure 7 presents a convenient way to visualize these various options.

#### 4.1. Full Custom VLSI Design

The design of a complex LSI or VLSI chip would allow a large number of possible procedural operations. This *full-custom* design is the least restrictive possibility by allowing the designer complete freedom. The major disadvantage of full custom design lies in the enormous cost and the long design time required. Nevertheless, most commodity items such as microprocessors use this method as it offers the best performance (speed, area, etc.).

A representative example for full-custom design is given in the next figure (Fig. 8). It shows a chip photograph of the  $\mu$ VAX II<sup>†</sup> interface chip.

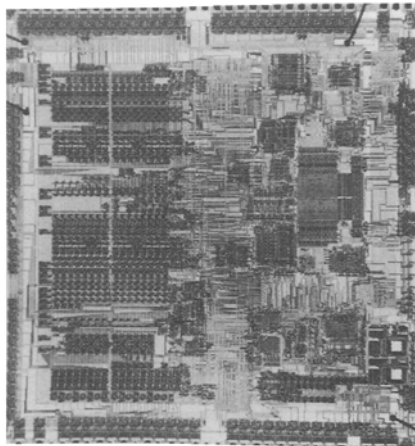


Fig. 8 Microphotograph of the  $\mu$ VAX-II interface chip. [6]

Implemented with a double-level metal,  $3\mu\text{m}$  silicon-gate NMOS process, the chip contains over 24,000 transistors that dissipate 3.5W of power. The chip performs three major functions: data transfer, bus control and error handling.

#### 4.2. Gate Arrays and Sea-of-Gates

On the other side of the spectrum are the gate array technologies. The principle of the gate array rests on the use of a prefabricated wafer that contains a regular and fixed arrangement of logic gates. In a final process step, a particular digital circuit is implemented by connecting a number of these cells with metal wires. Figure 9 shows a schematic picture of a gate array layout.

<sup>†</sup>  $\mu$ VAX II is a trademark of Digital Equipment Corporation

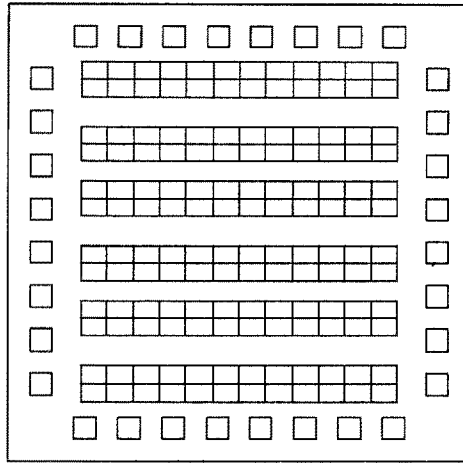


Fig. 9 Schematic drawing of a gate array chip

The prefabricated so-called "master" chip is typically mass produced and stocked. In CMOS technology, master chips are available with a complexity up to 20k gates. The advantage of the gate array technique lies in the very fast turn-around time (between 10 and 20 days), and the low cost. The disadvantages are mainly given by the area penalty due to the regularity and the low speed performance.

During the last few years, the gate array concept has been extended from the original basic cell approach to allow the incorporation of other circuit structures such as memories. This technique, called *Sea-of-Gates* or *Flexarray*, shows large promise for future VLSI chips. In the following, a few examples of this technique are presented.

Figure 10 shows the chip photograph of a control processor built with medium size gate array (4k gates) that also contains more than 2k of configurable memory. [7]

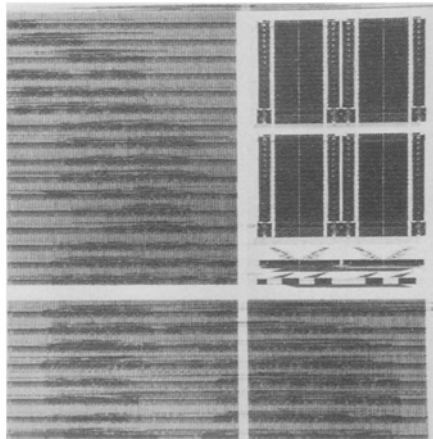


Fig. 10 Control processor implemented in a gate array with configurable memory. [7]

The basic memory array of this chip consists of eight  $32 \times 9b$  static memories. Any number of these eight independent memories can be combined into a larger memory by changing the metallization pattern. For example, four  $32 \times 9b$  memories could be changed into an  $128 \times 9b$ ,  $64 \times 18b$ , etc. memory. The basic gate array logical cells consist of a two-input NAND and a four-input NAND.

Another example of the sea-of-gates methodology is shown in Fig. 11.

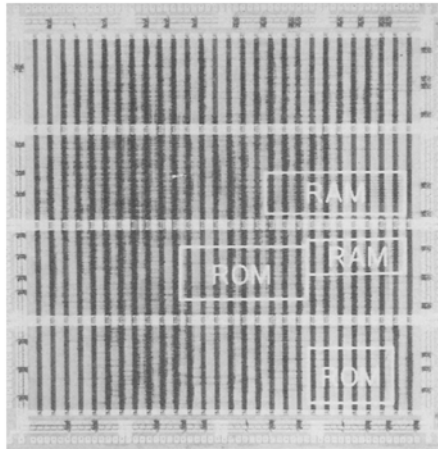


Fig. 11 16b microprocessor implemented using a CMOS gate array with configurable RAM and ROM [8]

The picture shows a 16b microprocessor implemented in a gate array chip was configurable RAM and ROM blocks. The gate array contains 864 columns by 28 rows of basic cells with 190 I/O buffers and 24 pads for power supplies.

The last and most impressive example of the sea-of-gates approach is presented in Fig. 12. [9] A 240K transistor CMOS gate array with flexible allocation of memory and channels has been developed, which has the following features;

1. The basic cells are designed to accommodate both logic unit cells and memory cells efficiently.
2. Wiring channels can be allocated freely in discrete units wherever necessary.
3. Logic and memory blocks may be placed in arbitrary positions on the cell array.

The examples shown illustrate the power of the sea-of-gates approach. The combination of prefabricated VLSI master slices with fully automated design tools offers many advantages over full custom methods, such as very fast turn-around time and ease of implementation.

#### 4.3. The Standard Cell Approach

While gate arrays use a prefabricated master, standard cell implementations need a complete process cycle for realization. Standard cell techniques are based on the use of a library containing several hundred of combinational and sequential logic cells. These cells are predesigned in a particular technology and thoroughly characterized through the use of circuits simulation tools. Compared to the gate array approach, this large library of cells gives the designer the increased freedom for efficient circuit and system design.

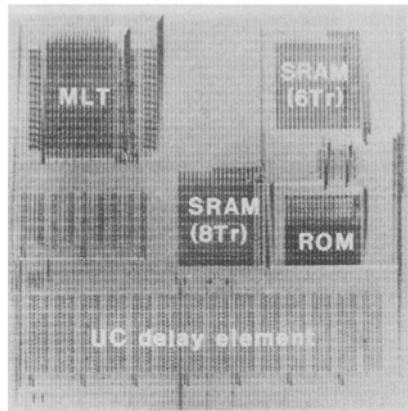


Fig. 12 Test Chip using a 240K gate array [9]

Figure 13 gives an indication of a typical standard cell layout.

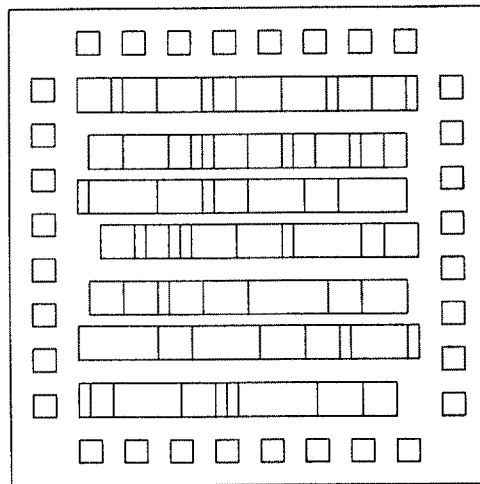


Fig. 13 Layout of a standard cell chip

The starting point for the design is the logic diagram of the circuit to be realized (the netlist). Through the utilization of existing library cells, this logic diagram is readily converted into a layout. The cells are placed in multiple rows and interconnected. In modern design tools, this placement and routing of cells have been fully automated. [10] The larger cell library and the placement/routing concept lead to "better" layouts if compared to the possibilities offered by the gate array approach.

#### 4.4. Macrocells

Apart from full custom design, the highest degree of flexibility is provided by the macrocell approach. The appealing idea behind the macrocell methodology is the inherent use of hierarchy. The conceptual idea of this technique can be seen in Fig. 14.

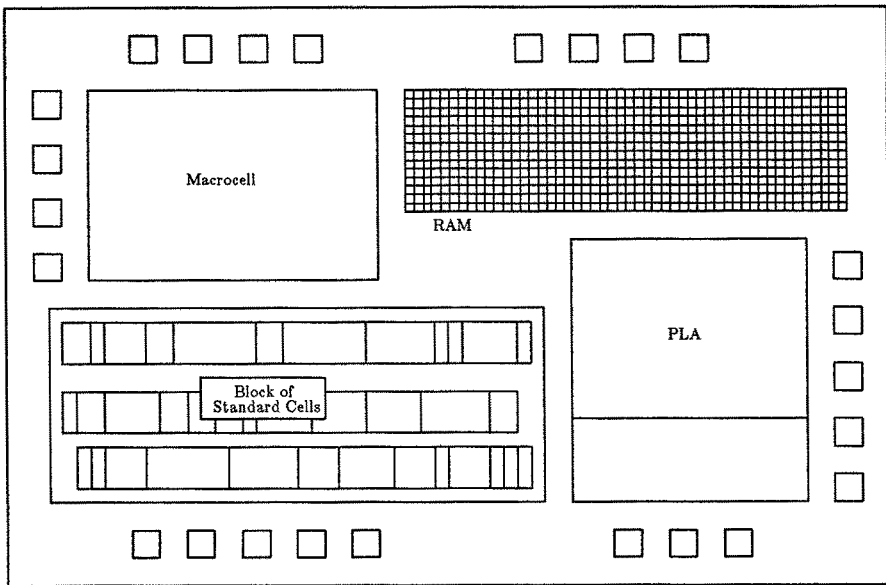


Fig. 14 The macrocell approach [11]

The basic building blocks are macrocells that can be of any rectangular shape. In addition, these subblocks can be designed by other design techniques such as standard cells, full custom design, etc. Furthermore, regular blocks such as PLAs, ROMs and RAMs are easily integrated in this concept.

Figure 15 illustrates this approach by showing the photograph of a 80b floating point processor chip containing a total of 150K transistors.

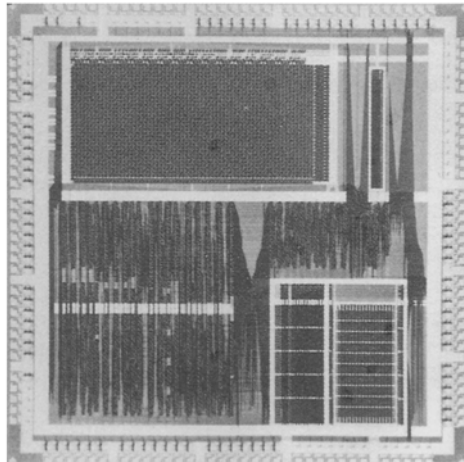


Fig. 15 Single chip 80b floating point unit

This chip contains a 64b multiplier, a 128w x 98b ROM, a 32w x 80b dual port register file and control logic. With the exception of the control part designed in a standard cell approach, all other pieces have been designed using a modular approach. The chip was laid out hierarchically (chip, block, cell). Block placement was executed using an interactive floor-plan editor. [12]. Highly regular blocks such as the ROM, the multiplier and the register unit were designed using the modular cell approach to achieve high integration density. The other blocks were constructed with 200 to 400 predefined standard cells to shorten turn-around time. Intra-block and inter-block routing were carried out simultaneously through a fully automated layout design program.

The macrocell approach is especially suited for the implementation of application-specific integrated circuits (ASICs). Figure 16 shows the micrograph of a dedicated single chip processor for a 32b Adaptive Differential Pulse Code Modulation CODEC implemented in  $2.5\mu\text{m}$  CMOS technology. [13] This processor consists of a data ROM, data RAM, instruction ROM, multiplier, ALU, normalizer/shifter and temporary register. About 60000 transistors have been integrated. The layout for the ROMs, RAMs and the multiplier were done manually, and the topological connections were verified using a CAD tool. The remaining part of the processor (about 40% of the total area) was designed using an automatic layout program based on the standard cell approach.

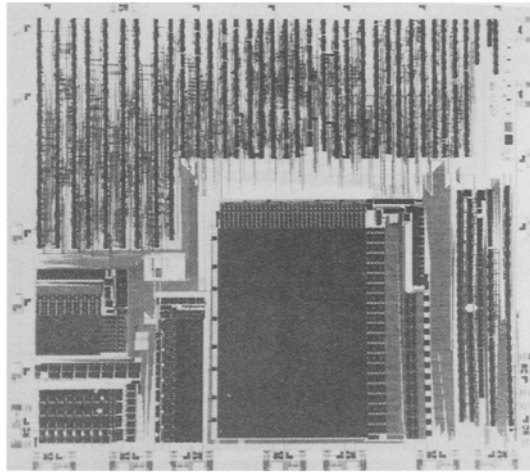


Fig. 16 Signal processor photograph [13]

## 5. Conclusions

The emergence of VLSI chips has brought the development of several distinct design methodologies. In this paper, the basic principles of these methodologies were reviewed. Representative examples of existing design have been chosen to illustrate the power of the various methods.

While each of these methodologies has its advantages and disadvantages, a certain trend has been established that combines the positive points in one design. The Sea-of-Gates and the Macrocell approaches, together with the use of automatic placement and routing tools, promise enough flexibility that future VLSI designs with several million chips can be successfully performed.

## References

1. FICHTNER, W. AND MORF, M., *VLSI CAD Tools And Applications*, Kluwer Academic Press, Boston, 1987.
2. KAHNG, D. AND ATALLA, M. M., *Silicon-silicon dioxide field induces surface devices*, 1961. IRE Solid-State Device Res. Conf., Carnegie Institute of Technology
3. W. FICHTNER, E. N. FULS, R. L. JOHNSTON, R. K. WATTS, AND W. W. WEICK, "Optimized MOSFETs for Subquartermicron Channel Lengths," *IEEE International Electron Devices Meeting Technical Digest*, pp. 384-387, 1983.
4. NOYCE, R. N. AND HOFF, M. E., "A History of Microprocessor Development at Intel," *IEEE Spectrum*, p. 8, 1981.
5. BEYERS, J. W., DOHSE, L. J., FUCETOLA, J. P., KOCHIS, R. L., LOB, C. G., TAYLOR, G. L., AND ZELLER, E. R., "A 32-Bit VLSI CPU Chip," *IEEE J. SC*, vol. 16, p. 537, 1981.
6. SCHUMANN, R. AND PARKER, W., "A 32b Bus Interface Chip," *ISSCC Technical Digest*, p. 176, 1984.
7. SANO, T., MATSUKUMA, S., HASHIMOTO, K., OHUCHI, Y., KUDO, O. AND YAMAMOTO, H., "A 20ns CMOS Functional Gate Array with a Configurable Memory," *ISSCC Technical Digest*, p. 146, 1983.
8. UEDA, M., SAKASHITA, K., YONEZU, R., FUJIMURA, T., ARAKAWA, T., ASAI, S. AND KURAMITSU, Y., "A 1.5 $\mu$  CMOS Gate Array with Configurable RAM and ROM," *ISSCC Technical Digest*, p. 126, 1985.
9. TAKAHASHIM H., SATO, S., GOTO, G. AND NAKAMURA, T., "A 240K Transistor Gate Array with Flexible Allocation of Memory and Channels," *ISSCC Technical Digest*, p. 124, 1985.
10. HORBST, E., NETT, M. AND SCHWARTZEL, H., *VENUS - Entwurf von VLSI Schaltungen*, Springer-Verlag, Berlin, 1986.
11. WECKER, T., *Semi-Custom Design Systems*, in Logic Design and Simulation, (ed. by E. Horbst), 1986.
12. UEDA, K., KITAZAWA, H. AND HARADA, I., "CHAMP: Chip Floor Plan for Hierarchical VLSI Layout Design," *IEEE Trans. CAD*, vol. 4, p. 12, 1985.
13. SATO, M., ISHIKAWA, Y., NISHITANI, T., KATO, T., SAITA, H. AND AOKI, Y., "A Single Chip Signal Processor for CCITT Standard ADPCM Codec," *ISSCC Technical Digest*, p. 192, 1985.

# EMBEDDED SYSTEM DESCRIPTION USING PETRI NETS

Wolfgang Reisig

Gesellschaft für Mathematik und Datenverarbeitung

Institut für Methodische Grundlagen

5205 Sankt Augustin, Germany

## Abstract

By a number of examples it will be demonstrated how Petri nets can be applied to design and to analyze embedded systems. Principles of Petri nets as a design method are discussed. Various application areas are glanced, and some relationships to other methods are sketched.

## CONTENTS

1. SOME ISSUES OF EMBEDDED SYSTEMS DESIGN
  - 1.1 Essentials of a Design Method
  - 1.2 An Example
2. SOME ASPECTS OF NET BASED SYSTEMS DESIGN
  - 2.1 Causality Based Executions
  - 2.2 Conflicts and their Resolution
  - 2.3 Coordination Mechanics
  - 2.4 Managing Individual Objects
3. SOME ANALYSIS TECHNIQUES
  - 3.1 Design Control
  - 3.2 S-Invariance
  - 3.3 Further Analysis Techniques
4. SOME APPLICATION AREAS
  - 4.1 Process Control
  - 4.2 Data Flow Systems
  - 4.3 Miscellaneous
5. SOME RELATIONSHIPS TO OTHER METHODS
  - 5.1 Petri nets and SADT
  - 5.2 Petri nets and Jackson Structured Programming

CONCLUSION

REFERENCES

## 1. SOME ISSUES OF EMBEDDED SYSTEMS DESIGN

In this introductory chapter we discuss some aspects of system design that are essential for embedded systems design, and discuss by help of an example the first steps in the design of embedded systems.

### 1.1 Essentials of a Design Method

The literature on system design techniques includes a vast amount of papers on different methods to specify complex systems. Most methods stress certain aspects of the development process, or turn out useful mainly for distinguished application areas. Petri nets turn out especially useful for so called computer-embedded systems.

It is characteristic for such systems that the interfaces between computers and their environments are not trivial. The environments may include various technical components (as e.g. in industrial control systems) or they may be more human-oriented (as e.g. office systems). In general the environments consist of some technical devices that are used or controlled by humans.

To understand the role of a computer in such systems, it is essential that one has a precise comprehension of the behaviour of its environment. There must be a formal model not only of the involved computers behaviour, but also of the environment's behaviour. To prove the correctness of software components, it is extremely useful if the software and the system environment make use of the same description technique.

Petri nets provide such a description technique. The general concepts of "events" and "conditions" can be applied in software design as well as in the specification of the role of human users or operators. Hierarchical design principles yield a proper design technique.

Petri nets is a collective term that, in the course of time, has come to designate a large number of system models, procedures, descriptive patterns and techniques related to one another in the sense that they are all based on the same specific principles. In addition, there are systematic transitions between them which, themselves, make up part of Petri net theory.

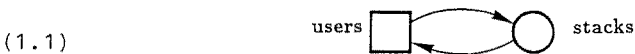
The relevant aspects of system design with Petri nets, as well as some analysis techniques will be introduced and discussed by help of examples. Our aim is to give the reader a feeling of what can be done with nets, rather than a full-fledged formal theory. The graphical representation of Petri nets is so intuitive that the reader will quickly

get an intuitive understanding of the general and formal background.

For a more systematic, yet intuitive introduction of Petri nets we refer to [Reisig 85b]. A formal introduction gives [Reisig 85a], and many other introductory papers and books.

## 1.2 An Example

The following example is taken from [Reisig 85b]. It is intended to give a first impression of the use of Petri nets in the course of designing a system. It concerns the initial phases in the system development process, i.e. when system requirements are often available only with the impreciseness and incompleteness characteristic of every day language. The idea is to gradually clarify, complete, and eventually formalize these requirements, and to develop a dynamic model which meets these requirements. Taking the organization of a library as an example, let us begin with the most basic meaningful structure of a library. The users of the library have access to a stock of books. Thus, we are dealing here with two components. It would seem obvious enough to describe one of the components (the stock of books) as being "passive", and the other (the users) as being "active". If we represent the passive component with a circle and the active component with a box we have the configuration shown in Figure (1.1). The arrows between the components indicate the flow of objects and information.

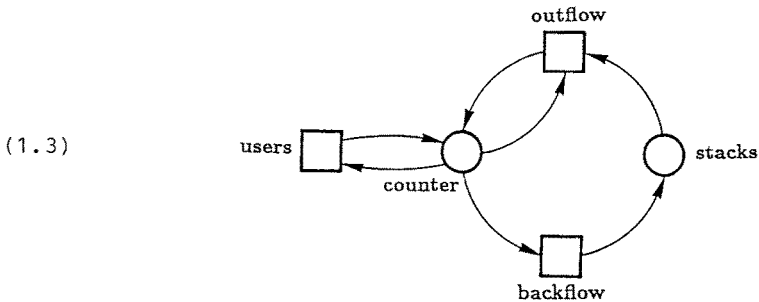


In large libraries a user may not have free access to the stacks. Instead there may be one (or several) counters at which users are waited on by library staff. Intuitively we understand which components are passive (the counters) and which are active (library staff). Based on this information, we can move on to the organizational structure indicated in Figure (1.2).

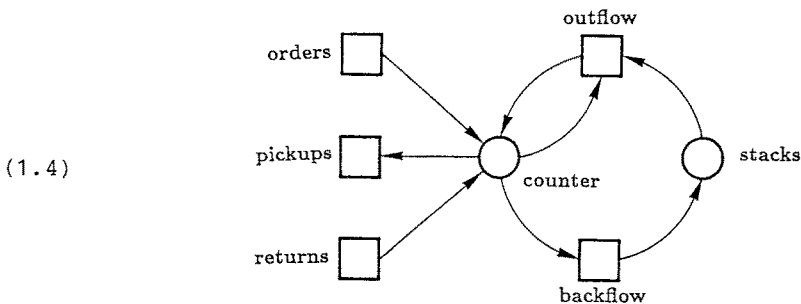


A large volume of book borrowing requires organization. We first of all need to distinguish the lending (outflow) and return (backflow) of books. In Figure (1.3) assume that before books can be borrowed they

first of all have to be ordered. In other words, there has to be a flow of information from the borrower to the lender before a given book will be transferred from the lender to the borrower.



In this connection, it would seem useful to identify the different ways in which the borrower interacts with a library, i.e. when he orders a book, when he picks it up, and when he returns it (cf. Figure (1.4)).

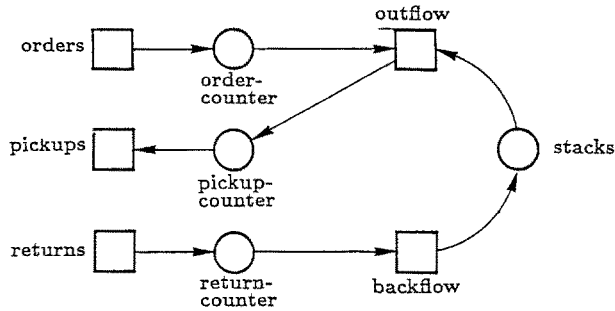


In order to be able to deal with large numbers of people at one time, it would seem sensible to establish separate counters for ordering, picking up and returning books (Figure (1.5)). The library keeps a record of the books borrowed, in the simplest case by using index cards that are stored with the books in the stacks. Whenever a book is borrowed, its card is placed in a borrowed-book file. When the book is returned, the relevant card is taken out of the file and returned to the stacks along with the book it belongs to.

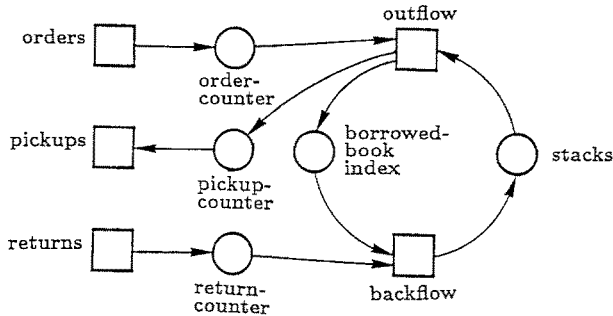
Figure (1.6) adds a borrowed-book index to the system. (This would only be the first step if we wanted to plan the computer-aided reorganization of a manually administrated library.)

At this point we should focus on a central principle of system design with nets, i.e. that of systematically moving from nets made up of channels and agencies, such as the one we have just dealt with, to nets

(1.5)

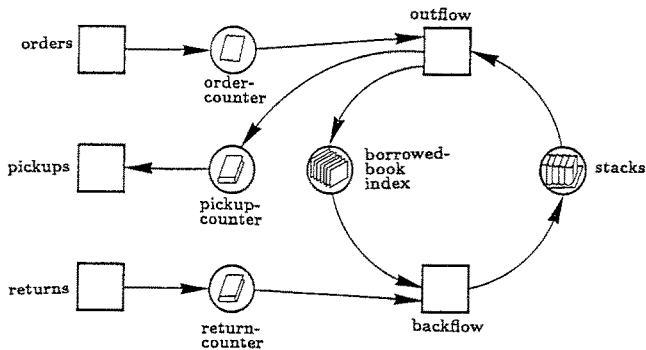


(1.6)



that model dynamic behavior. Figure (1.7) shows that the channels contain concrete objects, i.e. an order form (filled out) is at the order counter, a book is at the pickup counter waiting for the person who ordered it, and at the return counter there is a book that has been returned but not yet put back on the shelf.

(1.7)



The books in the stacks and the cards in the borrowed-book index are also indicated. The agencies can now redistribute the objects, acting in accordance with rules formulated by the system designer. Order forms are accepted by the lending agency, books are retrieved from the stacks and brought to the lending counter. Borrowers pick up the books they have ordered from the lending counter. Returned books are put back in the stacks.

## 2. SOME ASPECTS OF NET BASED SYSTEM DESIGN

In this chapter we present typical and distinguished ways of thinking in net theory. We again present the material mainly via examples.

Single runs (executions) of distributed systems deserve our special attention. This topic is discussed in Section 2.1. Then we turn to problems of synchronization and coordination in embedded systems, and discuss a railway control system. Conflict, or nondeterminism and fair resolutions of such situations is the topic of Section 2.3. Finally we shall discuss the proper treatment of individual objects as tokens in nets, which has already been outlined in Section 1.2.

### 2.1 Causality Based Executions

The way of thinking about single runs (executions) of distributed systems is fundamentally different in nets, in comparison with other system models.

In an abstract setting, a single run of any system is usually conceived as a sequence

$$(2.1) \quad s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \xrightarrow{t_3} \dots$$

of states  $s_0, s_1, \dots$  and transitions  $t_1, t_2, \dots$ . This is motivated by the assumption that any system run proceeds in time. The states  $s_i$  are system states of certain points in time. The increase of the index  $i$  corresponds to the forward direction of time. Sometimes one only considers the sequence  $s_0 s_1 s_2 \dots$  of states; in other contexts one concentrates on the transition sequence  $t_1 t_2, \dots$ ; but generally the above model (2.1) appears so very intuitive, that it seems needless to discuss its justification. All operational models of languages such as CCS, CSP, Ada and specification techniques for nonsequential systems, such as temporal logic, rely on the execution model (2.1). In an abstract setting, representation techniques which specify such sequences, are usually denoted as "transition systems" (cf. the various contributions in [Apt 1985]).

In the net theoretical approach it becomes evident that such sequences mistake essential aspects of the behaviour of distributed systems. A thorough look at the behaviour of distributed systems brings a lot of hidden assumptions to light, which will be discussed in the following.

We start with a very elementary example of a producer/consumer system

as follows: There is a producer that alternately produces items and delivers them to a buffer. The consumer may remove items from the buffer and consumes them. The buffer itself is assumed unlimited. Figure (2.2) shows this system.

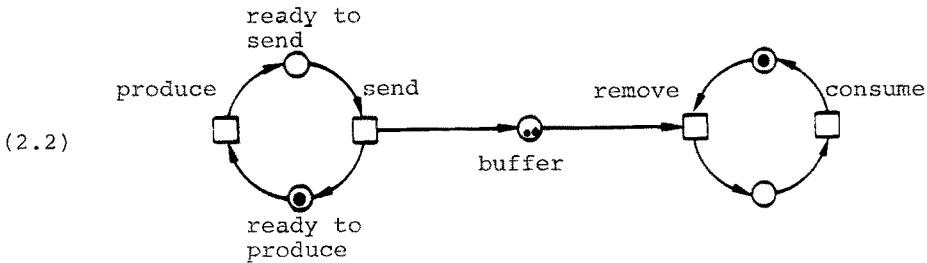
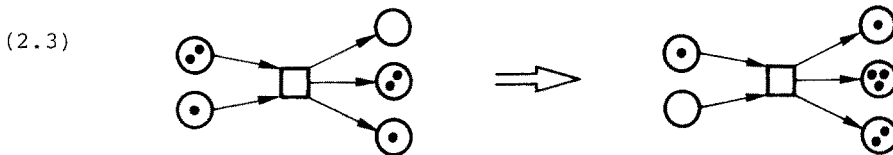


Figure (2.2) shows a net with a quite simple rule of dynamic change. To give an example, "send" is an event that may occur in case the place "ready to send" holds a token (represented by a black dot). The occurrence of "send" then removes this token from "ready to send", places a token to "ready to produce", and coincidentally augments a further token to the buffer. In the situation shown in Figure (2.2), "send" may occur only after the occurrence of "produce".

The general occurrence rule for events is outlined in Fig. (2.3). We assume this rule be sufficiently clear by this example and do without a formal definition.



The behaviour of the system (2.2) is a combination of repeated occurrences of the four involved events; viz. of "produce", "send", "remove", and "consume". (We abbreviate these events in the following by their initial character, and write p,s,r and c, respectively.) It is essential that the behaviour of the system (2.2) will not be formulated in terms of a time scale, but exclusively in terms of causal dependencies.

We first turn to the producer, whose behaviour is given by the sequence

(2.4) 
$$r \rightarrow c \rightarrow r \rightarrow c \rightarrow r \rightarrow \dots$$

of alternate production and sending of items. Read from left to right,

the  $i$ -th occurrence of "p" denotes the  $i$ -th occurrence of the event "produce". The occurrences of "s" likewise denote the corresponding occurrences of the event "send" of (2.2). Arrows denote the direct predecessor/successor relation. Hence the  $i$ -th occurrence of "produce" precedes the  $i$ -th occurrence of "send", which in turn precedes the  $i+1$ st occurrence of "produce". The occurrences of "receive" and "consume" are likewise ordered:

$$(2.5) \quad r \rightarrow c \rightarrow r \rightarrow c \rightarrow r \rightarrow \dots$$

The point to be discussed now is the combined behaviour of the producer and the consumer. This roughly should be a combination of (2.4) and (2.5). It is clear that the reception of the  $i$ -th item must be preceded by its being sent. So the overall behaviour of (2.2) is given by (2.6).

$$(2.6) \quad \begin{array}{ccccccc} p & \rightarrow & s & \rightarrow & p & \rightarrow & s \rightarrow \dots \\ & & \searrow & & \searrow & & \searrow \\ & & r & \rightarrow & c & \rightarrow & r \rightarrow \dots \end{array}$$

(2.6) precisely describes the causal dependencies among the event occurrences of the system (2.2). Arcs  $a \rightarrow b$  in (2.6) should be read "a is direct predecessor of b" or "a is one of the last event occurrences necessary for the occurrence b". The transitive closure of  $\rightarrow$ , intuitively the sequences of arcs  $a_0 \rightarrow a_1 \rightarrow \dots \rightarrow a_n$ , describe the before-after relation, conceived however in a causal, not in a temporal sense. This concept has a number of implications, some of which are discussed in the following:

- (a) (2.6) is the only run of the system (2.2). This is because the system itself is deterministic. It includes no alternatives at all: Alternatives occur in nets as brached places . (2.6) does not contain such places.
- (b) Any attempt to conceive (2.6) as a sequence of states and transitions in the sense of (2.1) must fail. This remains valid even if we allow events of (2.2) to occur "simultaneously at the same time". We then would get orders such as

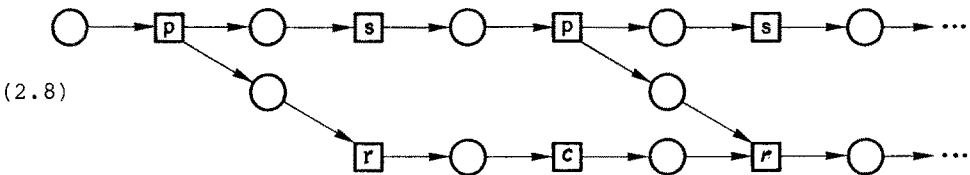
$$(2.7) \quad p \rightarrow s \rightarrow r \begin{array}{l} \nearrow p \\ \searrow c \end{array} \rightarrow s \quad \text{or} \quad p \rightarrow s \begin{array}{l} \nearrow p \rightarrow s \\ \searrow r \rightarrow c \end{array}$$

which still includes more ordering than (2.6).

- (c) It is not possible to supply global states to (2.6). In the presence of global states, each event occurrence is situated either before, or after each such state. Event occurrences between two neighbouring

global states are coincidently at the same time instance. This implies unordered, then denoting coincidence, be transitive. This is the case in (2.7), but not in (2.6).

Despite of (c) it is useful to distinguish local state components in between each pair of neighboured event occurrences of (2.6). To this end it is very intuitive to represent event occurrences as boxes, and local state components as circles, respectively. For (2.6) we obtain (2.8) as the final version of the "run" of (2.2).



The local states i.e. the circles in (2.8) correspond to the places of (2.2).

One may modify the system (2.2) and assume a finite capacity of the buffer. This impacts the runs of the system in such a way that the event occurrences will be ordered more strictly. For a buffer capacity of one item, the second send occurrence must be preceded by the first remove occurrence etc. We will however not go into details here.

It was the purpose of this section to demonstrate how adequate and precise the behaviour of distributed systems can be described by avoiding any notion of global state.

## 2.2 Conflicts and their Resolution

Conflict situations play a fundamental role in the behaviour of systems. They are characterized as situations which permit alternative courses of action. As discussed in the last section, such alternatives should not consist just in various orderings of independent event occurrences. The alternative should be an essential aspect of the underlying system instead. A necessary (but not sufficient) condition for conflicts is the presence of branching places .

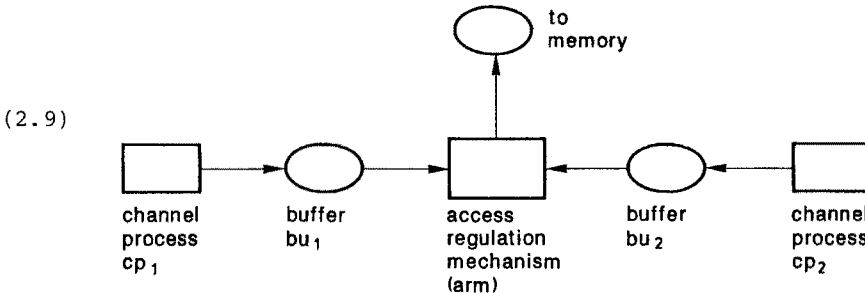
Conflicts may be resolved by information supplied from the environment to the system under consideration. As long as this environment is

not specified, one may assume conflict resolution by arbitration.

In a lot of applications, arbitration is perfectly acceptable as long as the system behaviour remains inside some prespecified "operation margins". Of this type is a large class of systems where a couple of conflicting parties compete for access to a shared resource.

We shall develop an access regulation mechanism for such systems. The conflicting parties may or may not be part of computers in an embedded system. As a concrete example we consider the orderly resolution of memory access conflicts among competing channel processes of modern high-performance computer system.

The channel processes are to autonomously transfer blocks of data between peripheral mass storage devices and the main memory. We discuss an extremely simplified situation of the case discussed in [Kluge, Lautenbach 82], where two such channel processes compete for access to the memory. The channel processes  $cp_1$  and  $cp_2$  are via buffers  $bu_1$  and  $bu_2$  linked to an access regulation mechanism (arm), which controls and interleaves the traffic from the buffers to the memory. This situation is sketched in Figure (2.9).



The behaviour of the channel processes can not precisely be predicted, but we assume  $cp_1$  requires on the average a quantity of  $a_1$  out of  $b$  accesses to the main memory ( $a_1 < b$ ). The rest of  $a_2 = b - a_1$  out of  $b$  accesses is granted to the channel process  $cp_2$ .

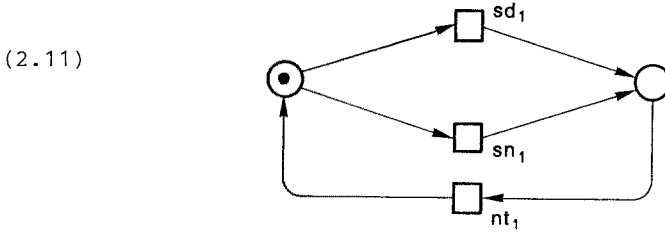
We furthermore assume a deviation of each channel process  $cp_i$  from its access quotient  $\frac{a_i}{b}$ . This means to assume naturals  $n_i$  and  $m_i$ , such that after  $k$  time units, the number  $q_i$  of accesses of  $cp_i$  is within the limits

$$(2.10) \quad \frac{a_i}{b} - n_i \leq \frac{q_i}{k} \leq \frac{a_i}{b} + m_i .$$

So we have to model

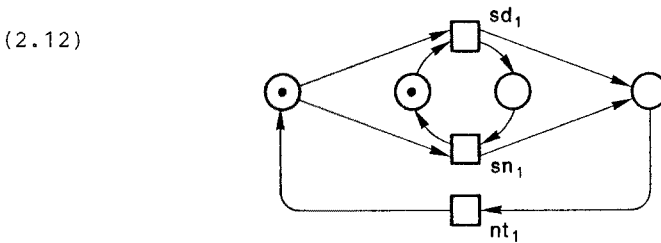
- the above assumed behaviour of the channel processes  $cp_i$ ,
- the buffers  $bu_i$  between the channel processes and the arm,
- the arm itself, guaranteeing a fair service for both channels.

To start with the channel processes, (2.11) shows the essential events of the channel process  $cp_1$ : It decides in each time unit

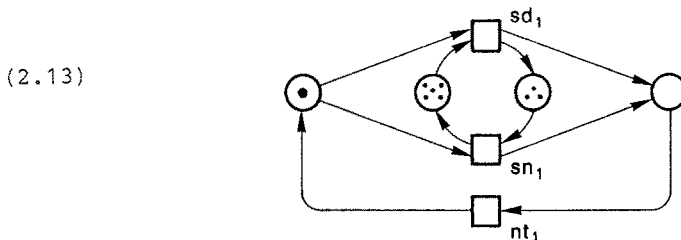


to send ( $sd_1$ ) or not to send ( $sn_1$ ) a data block to the buffer  $bu_1$ . The event  $nt_1$  introduces the next time unit.

(2.11) specifies arbitrary choice between sending and not sending data blocks. As an example for a deterministic conflict resolution, (2.12) specifies the strict alternation of  $sd_1$  and  $sn_1$ .

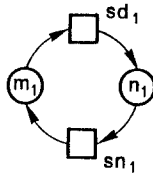


The more general case of the access quotient  $\frac{a_1}{b} = \frac{1}{2}$ , just as in (2.12), but with a deviation of, say,  $n_1=3$  and  $m_1=5$ , is shown in (2.13).



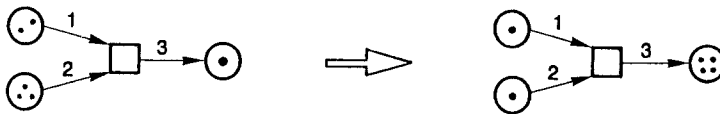
(2.12) and (2.13) already indicate the general technique of ordered conflict resolution: A regulation circuit

(2.14)



is constructed for conflicting events  $sd_1$  and  $sn_1$ . The deviations from the intended occurrence ratio is given by the number of tokens on the involved places. The ratio  $\frac{a_1}{b}$  itself is represented by arc weights. The arrows of the regulation circuit (2.14) will be weighted by integers. Such arc weights indicate the number of tokens "passing through the arc" upon occurrence of the involved event. An example should again outline the corresponding rule of change:

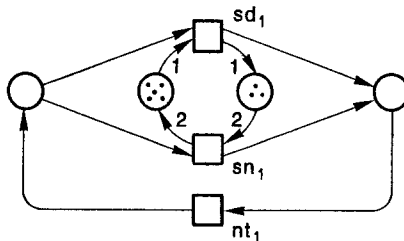
(2.15)



The so far applied unscripted arcs correspond this way to arc weights equal to one.

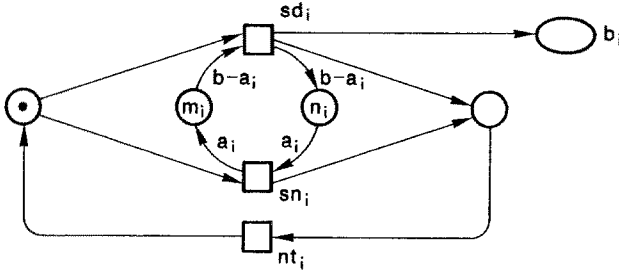
Returning to our channel process, we may assume a ratio of  $a_1 = 2$  and  $b = 3$  for (2.11). Together with the deviation of (2.13), we obtain for this case the net of (2.16).

(2.16)



This completes the specification of the channel processes  $cp_1$  and  $cp_2$ . To sum up,  $cp_i$  ( $i=1,2$ ) is characterized by the access quotient  $\frac{a_i}{b}$ , and the deviations  $n_i$  and  $m_i$  upwards and downwards, respectively. The general model is (2.17). (2.17) furthermore includes the buffer  $b_i$ ,

(2.17)

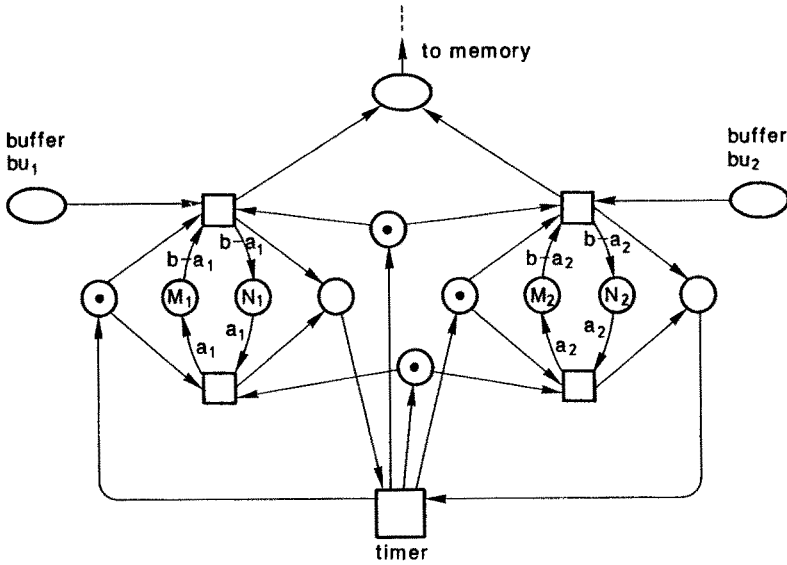


and the link from the channel process to the buffer. (2.17) is called a synchronization primitive. Such primitives are used in various contexts and environments.

We turn now to the access regulation mechanism, arm, which is to guarantee that in each time unit one data block is taken from one of the buffers  $b_1$  or  $b_2$ , and transferred to the memory.

We use synchronization primitives for the arm too. The synchronization primitives of the channel processes are more or less copied and linked by a common timer. Additional constraints guarantee that exactly one event transmits a data block. Figure (2.18) shows the final arm.

(2.18)

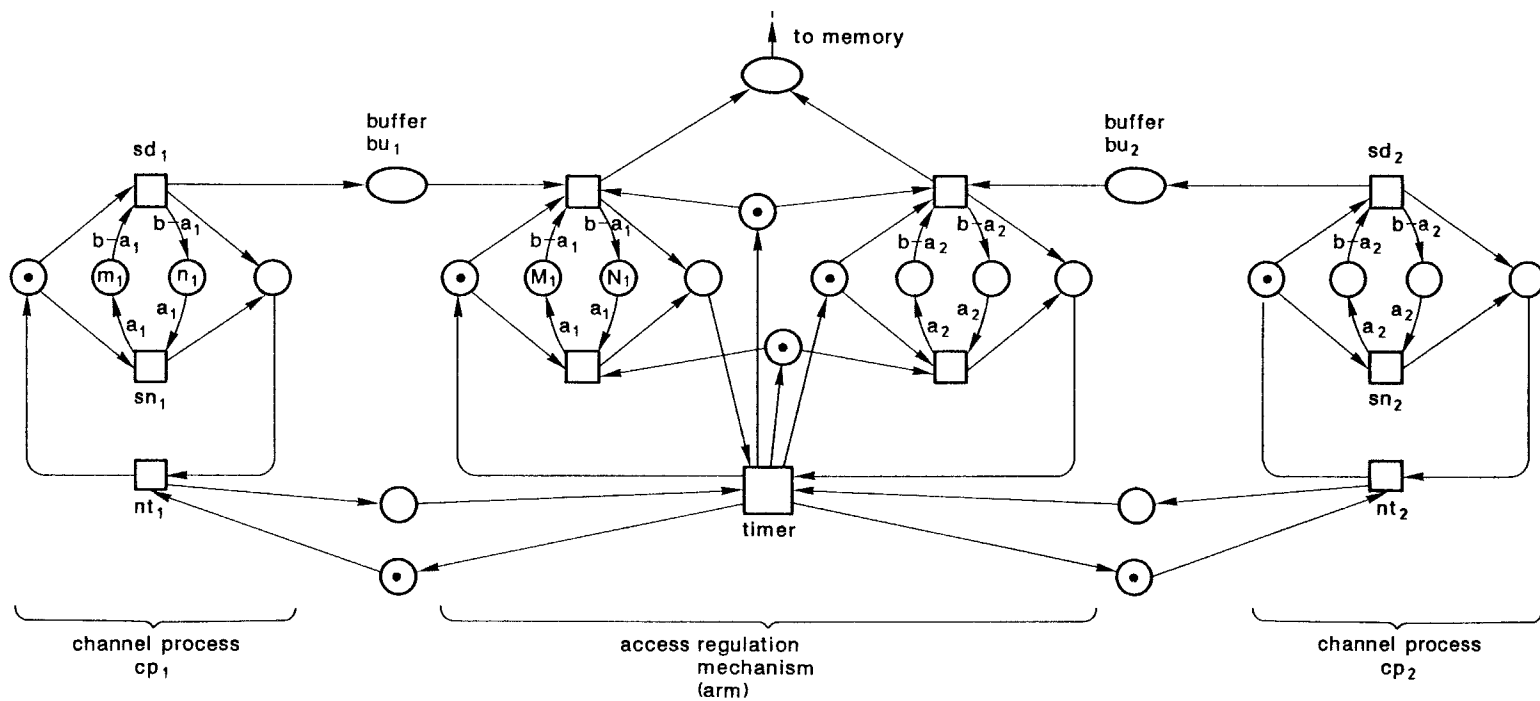


Notice that this access regulation mechanism operates in general not entirely deterministic. There may be cases where both buffers offer data blocks, and both components of the arm are capable for transmitting a data set. A conflict occurs in this case which may arbitrarily be solved by the implementation. It can also be used to implement priorities for one of the buffers, as long as access ratios and priorities are respected.

The ratios  $\frac{a_1}{b}$  and  $\frac{a_2}{b}$  are in the arm (2.18) fixed, and determined from the respective channel processes. The deviations  $M_1$  and  $N_1$  can however freely be chosen. We denote them by capitals in order to distinguish them from the corresponding deviations of the channel processes.

A final point to consider are the timers of the involved components. We assume local timers for the channel processes and the arm. This is reasonable only if we assume the timers be synchronized. We do not assume the timer of the arm and the events  $nt_1$  and  $nt_2$  (cf. (2.17)) occur coincidentally. (This would require the identification of the three events in one single event). It is sufficient, but also necessary, to harmonize them by synchronization circuits. So we come to the final version of our system, which is shown in (2.19)

If we assume the access ratios  $\frac{a_i}{b}$  and the deviations  $n_i$  and  $m_i$  ( $i=1,2$ ) be fixed, we have a number of parameters open for the disposition of the arm design. These are the deviations  $N_i$  and  $M_i$  inside the arm, and the



(2.19)

slack of the timers. These parameters can freely be chosen, but they influence the size of the buffers  $bu_1$  and  $bu_2$ . The buffer size has to increase with increasing values of  $n_i$ ,  $m_i$ ,  $|n_i - N_i|$ ,  $|m_i - M_i|$ , and growing slack of the timer synchronization. The most tight synchronization, i.e.  $n_i = N_i = 1$ ,  $m_i = M_i = 0$  and the timer synchronization as shown in (2.19) requires a size of  $bu_i$  of one item only. For other cases we leave the details to the reader.

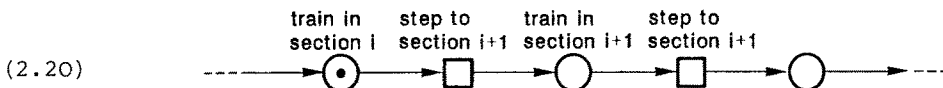
The purpose of this section was to study the notion of conflict and their proper resolution by synchronization circuits and synchronization primitives. Arbitration can be handled as long as some "operation margins" are observed. These operation margins can be chosen freely.

The role and modelling of time was a further subject of this section. Time occurs in technical systems always as a special form of signaling. The step to the next time unit corresponds to the occurrence of an event. Different timers with local scope should not be assumed as working absolutely identical, but should be synchronized. The degree of synchronization is left to the system designer. In this way we obtain a non-idealizing notion of time, subject to proper system modeling.

### 2.3 Coordination Mechanics

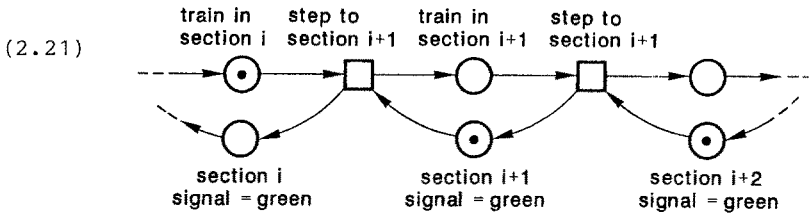
In this section we discuss the coordination of dynamic components in distributed and embedded systems. It will turn out that often it is superficial to assume and rely on, notions of time, even in systems which usually are denoted as time critical or time dependent. As noted above, in technical environments time occurs as signalling, e.g. as time out signalling, and should be modelled this way. Consequently one may concentrate on signaling, and avoid any notion of time.

As an example we discuss the coordination of trains running on one track. A track is usually subdivided into a sequence of sections, as Fig. 2.20 shows.

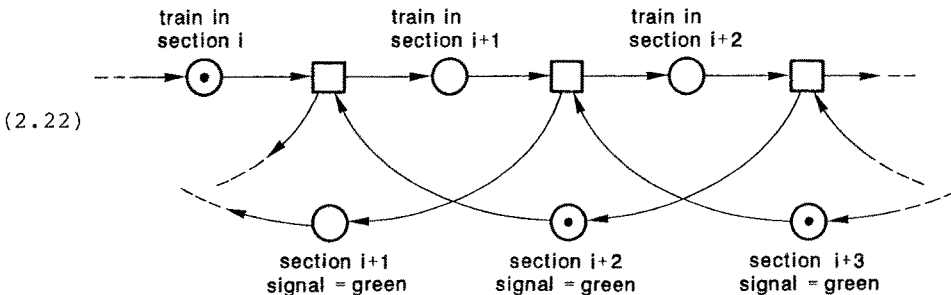


Trains are represented as tokens in the corresponding places. The case of emergency is given whenever two trains occupy the same section. This should be avoided by help of green/red signals. As a first idea we assume a signal for each section. The signals are to give permission to enter

free sections only. Figure (2.21) shows these signals.



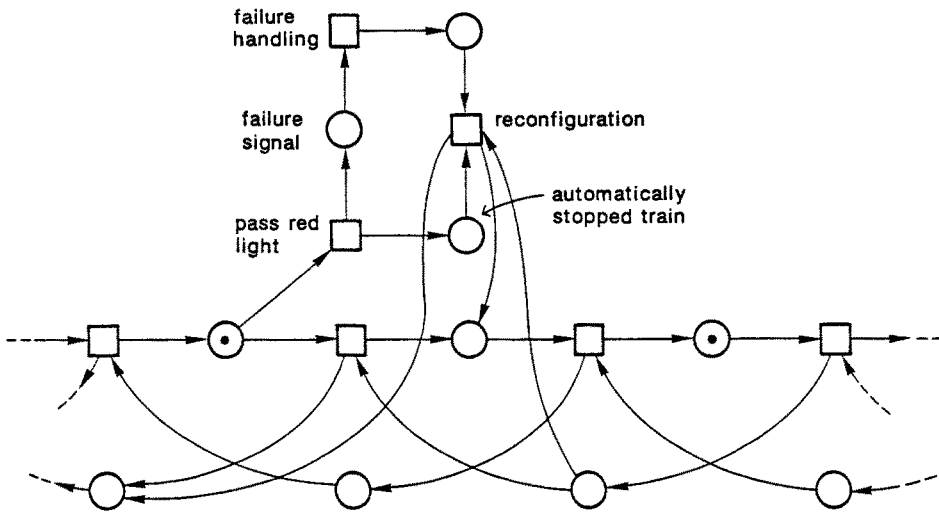
The signalling structure of (2.21) is not satisfactory. It is still dangerous to allow for trains in neighbour sections. A train by mistake may ignore a red signal and may enter an occupied section. Trains are automatically stopped upon passing a red signal, but sufficient space is required to really reach a stillstand. So we must make sure that upon entering section  $i$ , both sections  $i$  and  $i+1$  are free. This signalling scheme is given in Figure (2.22).



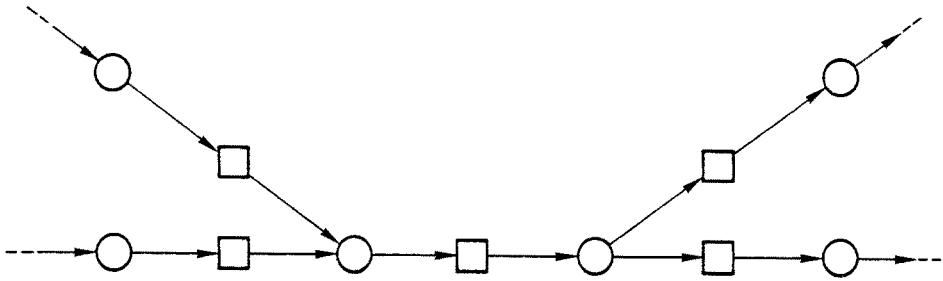
Trains and signals are treated on the same level in Figures (2.21) and (2.22). Both kind of items are modeled as tokens in the nets. Trains run from left to right; but there is furthermore a signal flow from right to left. These aspects are essential for proofs of correctness.

The case of emergency, which motivated (2.22), is explicitly modelled in Fig. (2.23). This Figure should be self-explaining by now. (In order not to overload the figures, we skip the inscriptions in (2.23)-(2.25).)

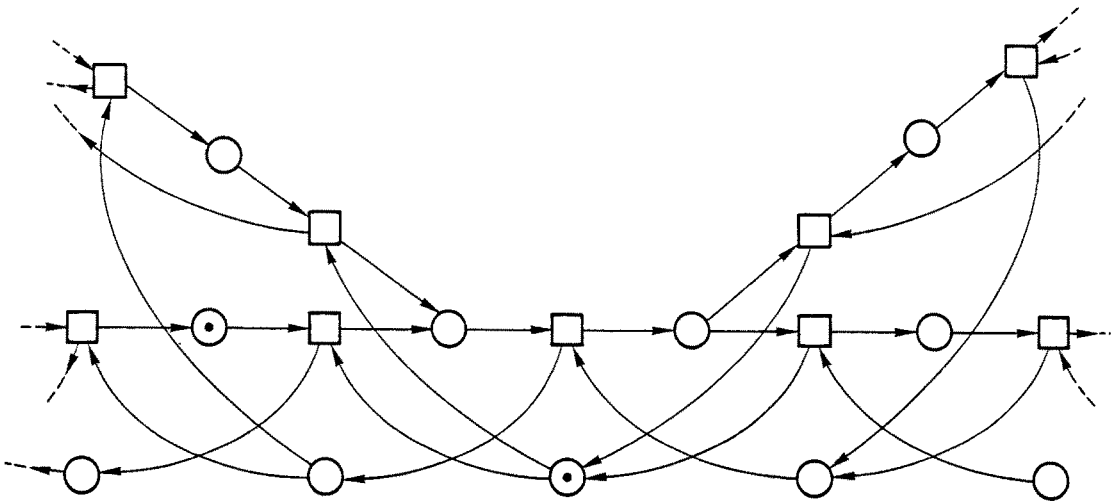
Our signalling scheme is flexible enough to cover the cases of branching as well as of cyclic tracks. A backward as well as forward branching track is shown in Fig. 2.24. Its full-fledged signalling scheme is just the superposition of all involved single tracks. It is presented in Figure (2.25).



(2.23)

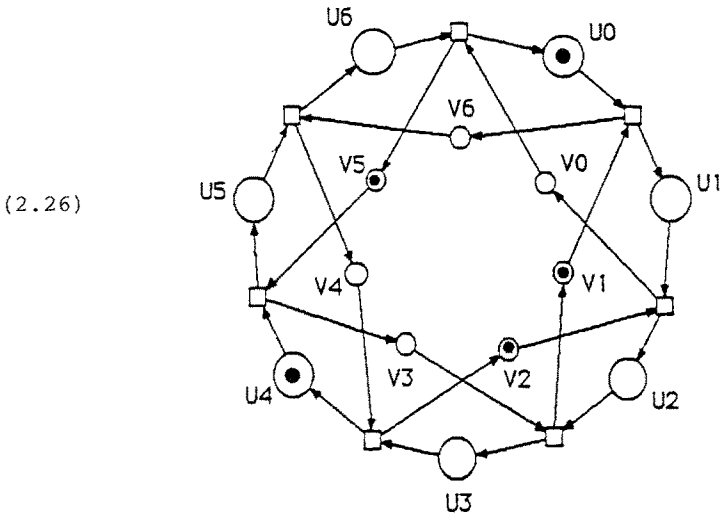


(2.24)



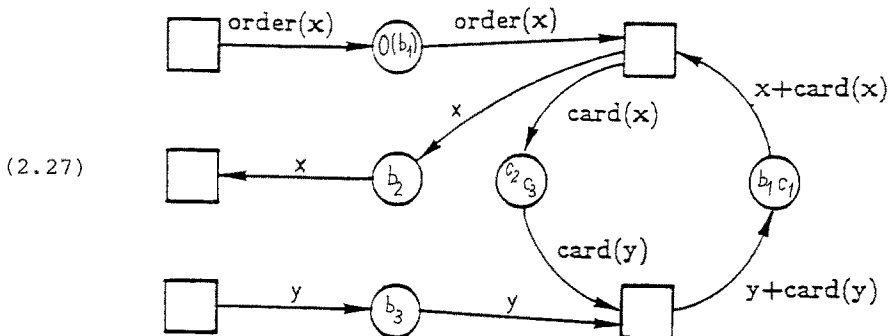
(2.25)

As an example for a cycle, Fig. (2.26) shows a cyclic trak system with two trains. It is a slight modification of an example in [Genrich 86].  $U_0, \dots, U_6$  denote "train in section 0, ..., section 6".



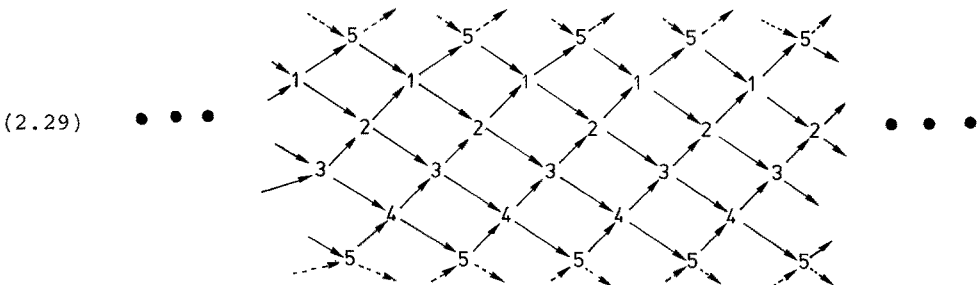
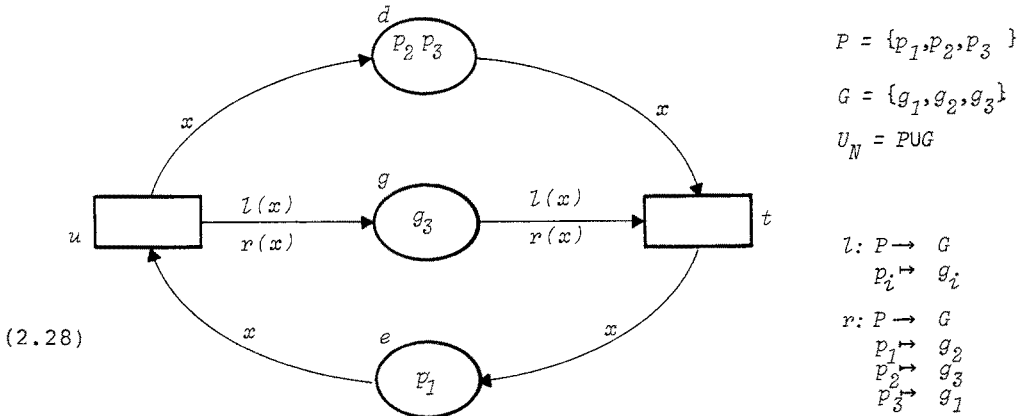
## 2.4 Managing Individual Objects

We saw already individual objects to occur as tokens of nets, in Chapter 1. Not just black dots, but books, index cards, and order forms have been moving round in the library system sketched in Figure 1.10. Fig. 2.27 represents its dynamic behaviour in a more formal setting.  $O(b_1)$  stands for an order for book  $b_1$ .  $b_i$  and  $c_i$  denote books and their corresponding index cards.



The arrows in Figure (2.27) bear inscriptions. They refer to the transition rule for such nets, which has the following significance here: The variables  $x$  and  $y$  stand for books in the library. For book  $x$ ,  $\text{order}(x)$  denotes a form on which book  $x$  has been ordered and  $\text{card}(x)$  denotes the index card for the book  $x$ . Events occur with respect to valuations of the variables, i.e. the variables are replaced with specific books. Thus, for example, the event "book ordering" can act by ordering the book  $x = \text{"New Approaches to the Formal Discription and Design of Embedded Systems"}$ . The result of this action is the presence of a filled-out order form on the order counter. Now the event "lending" can go into action, provided that the wanted book and its index card are available at the stacks. In this case they are removed from the stacks, the book is placed on the lending counter, and the card filed in the borrowed-book index.

Our next example is the standard paradigm of distributed systems, viz. the system of thinking and eating philosophers, as introduced in [Dijkstra 71]. As will known,  $n$  philosophers are sitting at a round table. Neighbour philosophers pairwise share a common fork. In order to get a meal, each philosopher needs both his left and his right fork. Figure (2.28) shows this system as a net with individual tokens.



This net is conflict prone to a high degree, because in order to get a meal, each philosopher has to argue with his two neighbours about the respective fork. So, many different runs of the system are possible, including those with starved philosophers who (may be due to conspiracy of the respective neighbours) never got both their forks coincidentally. As a concrete example we consider a well educated and cultivated circle of philosophers where neighbored philosophers alternately take their meals. The corresponding order of event occurrences is shown in Figure (2.29). This again is an example for a system execution that can not be represented as a run of a transition system in the style of (2.1). The case of five philosophers is considered. Each occurrence of a number,  $i$ , in (2.29) denotes a meal of the philosopher  $p_i$ . The dotted lines should pairwise be identified, in the obvious way.

### 3. SOME ANALYSIS TECHNIQUES

Some techniques for the construction of Petri net models have been presented in the last chapter. Now we turn to the analysis of such models. Our first topic shall concern the analysis of the process of constructing net models. This process includes systematic refinements and embeddings of intermediate results. Then we discuss a technique called S-invariance, which by mathematical methods supports the analysis of Petri nets. In its most elementary setting, sets of places with constant sums of tokens are characterized by help of solutions of linear equations.

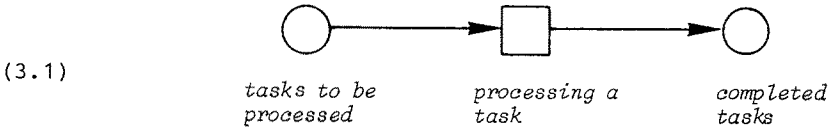
#### 3.1 Design Control

Petri Nets are not a "flat" method, i.e. real world components can be represented and systematically be developed on different levels of abstraction. An example for this technique has been discussed in Section 1.2 and - to a minor degree - in Section 2.2 with the step from Figure (2.9) to (2.19). In this way hierarchical models can be constructed by iterated refinement.

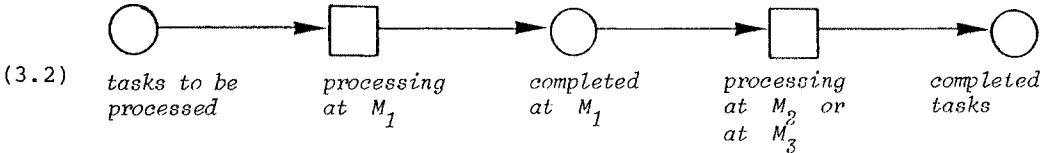
Refinements increase the knowledge about details of the modeled reality. The scope of concern remains constant by refinements. It may turn out during the process of system design that more components are to be modeled. This introduces a new dimension into the development process: the embedding of a net into a larger net. Refinements replace given components by more detailed descriptions, whereas embeddings supply new components.

We discuss the mentioned concepts again by an example, and study the very first steps towards the design of an industrial assembly line.

Fig. (3.1) shows the very

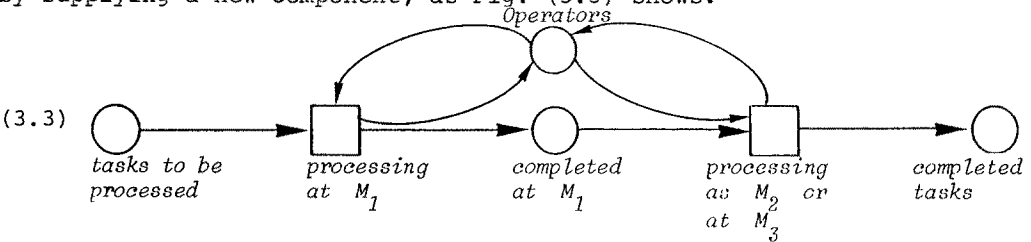


first approach to this line. The agency "processing a task" may be re-fined as Figure (3.2) shows. According



to this refinement, tasks are first processed at a machine  $M_1$ , and then at one of the machines  $M_2$  or  $M_3$ .

We now may have good reasons to also include the operators who handle the machines. This is represented as an embedding of Fig. (3.2), i.e. by supplying a new component, as Fig. (3.3) shows.



The design of real systems in general includes a large number of nets. The library example in Chapter 1.2 already includes no less than seven nets. If one thinks of further refinements of this system to specify a computer based system for the library organization and control, one may come up with an order of  $10^2$ - $10^3$  nets. Similar can be said about the above industrial assembly line.

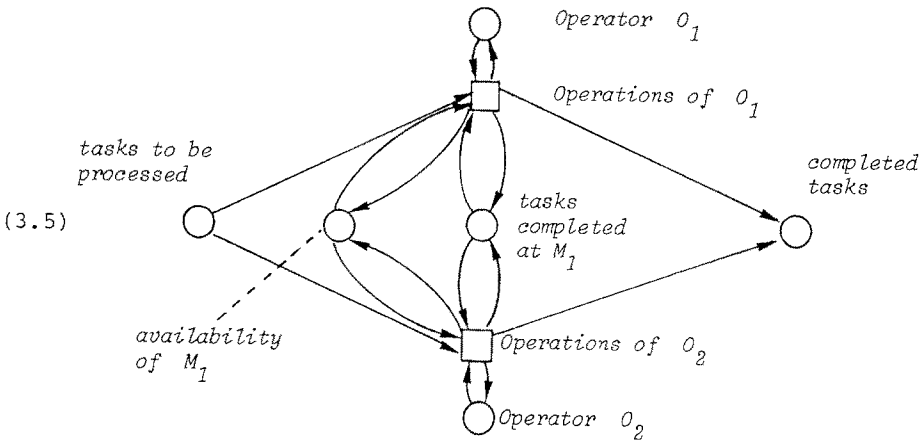
Distinguished techniques to handle such large collections of nets are mandatory. We introduce a graphical method that supports an overview of what has been achieved in any phase of the design process. It includes analysis techniques to discover certain inconsistencies and gaps within a design.

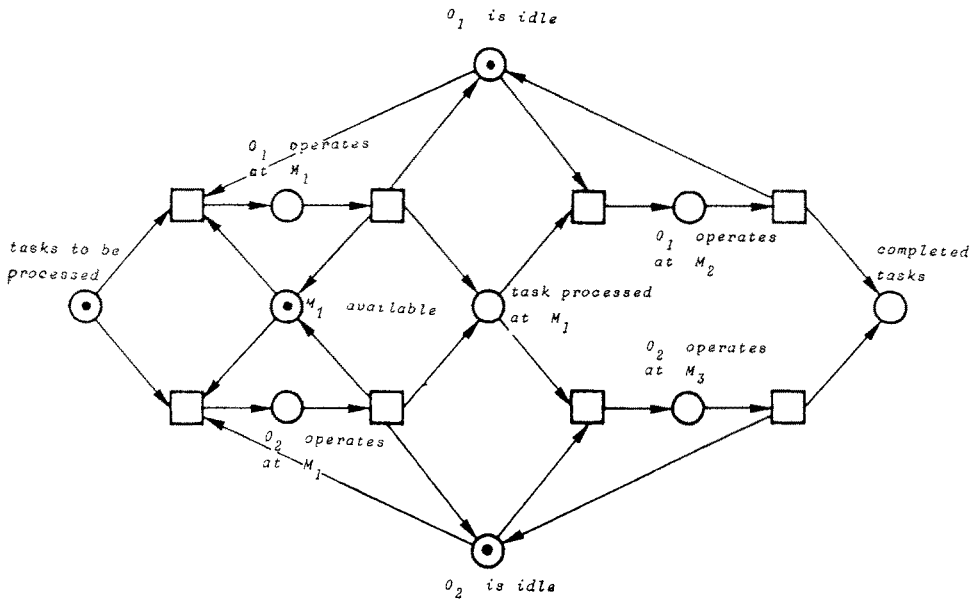
An overview of the relationships among given nets, as far as refinements and embeddings are concerned, is obtained by construction of a schedule. Each net is represented by an identifier (we shall apply the corresponding Figure number, as (3.4) illustrates for the so far

developed assembly line. Refinements are represented as simple vertical arcs ( $\downarrow$ ), embeddings are represented as double-lined horizontal arcs ( $\Rightarrow$ ).

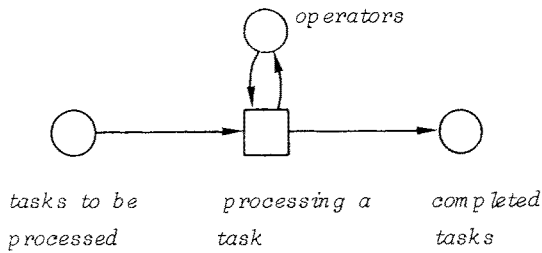


Such a schedule clearly supports the orientation about what has been achieved during the process of designing a system. Schedules furthermore provide necessary conditions for consistency and completeness of a designed system. In order to discuss these conditions, we extend the above example of an industrial assembly line as follows. Fig. (3.5) decomposes the assembly line from the operator's point of view. And Fig. (3.6) represents the system on a more refined level. The rule of change, as discussed in (2.15), applies to this net. We finally represent the system on a non-refined level, but include the operators in (3.7).



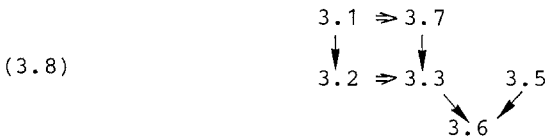


(3.6)



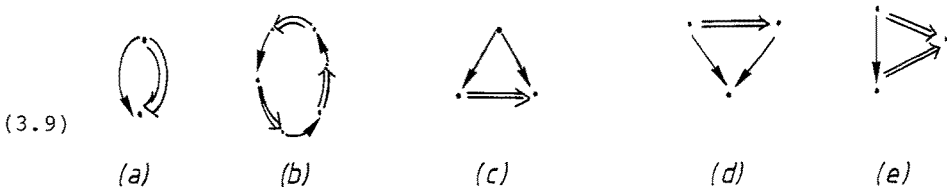
(3.7)

We complete the above schedule (3.4) and end up with the schedule



(3.8)

Certain forms of arrow sequences indicate that the design is certainly inconsistent. These forms are pointed at in (3.9). One should read the Figures in (3.9) in such



(3.9)

a way that one arrow of (3.9) corresponds in the schedule to a sequence of arrows of the same type.

Figure (a) in (3.9) shows a case where a net is refined (may be in a sequence of single refinement steps) as well as embedded (may be in a sequence of embedding steps), such that both operations yield the same net. The scope of represented topics would consequently remain unchanged, and would be extended as well. This is certainly an inconsistent specification. Figure (b) in (3.9) shows a case where a net may be re-gained after a sequence of refinements and embeddings. This certainly contradicts the idea of refinement and embedding. Figure (c) concerns a case where a net is refined in different ways. Different refinements should represent the same real components (in diverse details), hence one of them should not be a proper substructure of the other one. Figure (d) is the symmetric counterpart to (c), and the arguments given for (c) apply correspondingly. Figure (e), finally, shows the case of two nets where one is a refinement of the other one. They can be unified to one net, by augmenting further components to both of them. The original two nets then both occur as sections of the common embedding. According to Figure (e), this net would include a refinement of a part of itself.

Not each schedule without subgraphs as sketched in (3.9), is already consistent. Figure (3.9) only identifies a couple of inconsistency constructs which occur independently of the modelled reality.

It is quite obvious that the schedule of (3.8) is consistent in this sense.

The endeavour of a consistent system development process should - at least in principle - be just one net, which is most detailed and most comprehensive. This global property corresponds to a couple of local completion rules. The forthcoming Figures should be read as "... if a schedule contains nets A,B, and C, linked as indicated, then a complete schedule should contain the net D, too." As already in (3.9), arrows of the following Figures (3.10) to (3.12) correspond in schedules to sequences of arrows of the same type.

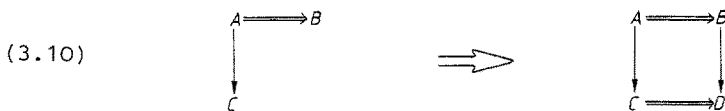
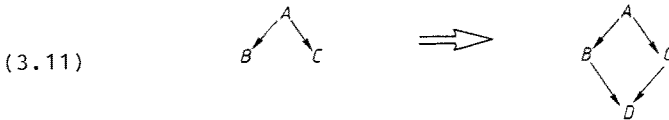
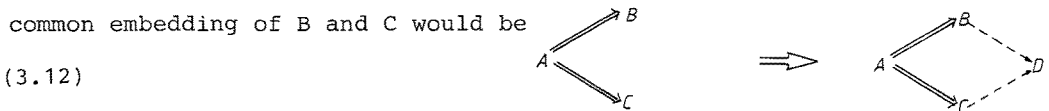


Figure (3.10) shows a case where a net A is a source for an embedding, B, as well as for a refinement, C. In a complete schedule, this is complemented by a net D, which is a refinement of B as well as an embedding of C. In some cases, D is unique. But we will not go into details here.

Next we consider the case where a net is refined in two different ways, as Figure (3.11) shows. In a consistent design, those two different refinements refer to one system; so it should be possible to join both of them by a further step of refinement in one net, D. This is what Figure (3.11) demands to exist in a complete schedule.



In Fig. (3.12) we consider the case in symmetric to Fig. (3.11): A net A is embedded into two different nets, B and C. The demand of a common embedding of B and C would be



too strict. There exist intuitively consistent schedules where in such a situation C is a refinement of B. In general we only demand the existence of a net D such that sequences of arcs of any type (in (3.12) represented by ---->) exist, which start at B and C, and end at D.

The schedule (3.8) meets all above requirements.

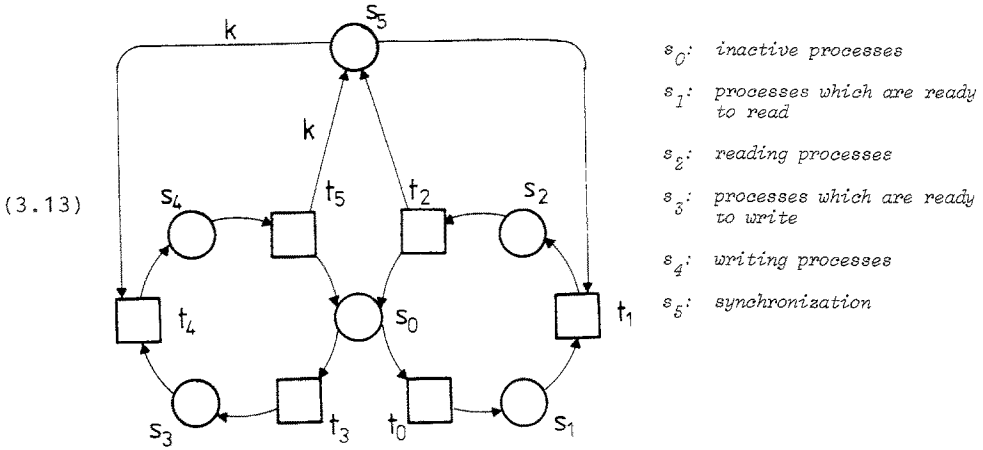
### 3.2 S-Invariance

We glance one of the specific analysis techniques for Petri nets in this section. Our aim is not to present the full-fledged calculus, but to give an example for the mathematical background and basis of net theory. We again do the job by help of an example.

As an example we again take a problem of shared resources. Suppose that  $n$  processes in an operating system are each allowed to access a buffer in reading or writing mode. To guarantee reliability, reading and writing access is restricted in the following way: When no process is writing to the buffer then up to  $k \leq n$  processes are allowed to read it. But writing access to the buffer is only permitted as long as no other process is reading or writing the buffer.

In Fig. (3.13), such a system of reader and writer processes is shown. Each process is in one of five states, represented by the places  $s_0, \dots, s_4$ . In the initial state all  $n$  processes are passive; hence  $s_0$  contains  $n$  tokens under the initial marking. The place  $s_5$  contains initially  $k$  tokens. This corresponds to the number of processes which are allowed to read the buffer concurrently.

With the technique of S-invariants it is possible to prove a number of properties of the system (3.13). S-invariants are obtained as solution of a system of homogeneous linear equations which in turn is derived



from the net under consideration. For this purpose to each net  $N$ , a matrix  $\underline{N}$  is constructed. The matrix of the net in (3.13) is given in (3.14). (Zero entries are omitted.)

|       | $t_0$ | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $i_1$ | $i_2$ | $M$ |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-----|
| $s_0$ | -1    |       | 1     | -1    |       | 1     | 1     |       | $n$ |
| $s_1$ | 1     | -1    |       |       |       |       | 1     |       |     |
| $s_2$ |       | 1     | -1    |       |       |       | 1     | 1     |     |
| $s_3$ |       |       |       | 1     | -1    |       | 1     |       |     |
| $s_4$ |       |       |       |       | 1     | -1    | 1     | $k$   |     |
| $s_5$ |       | -1    | 1     |       | - $k$ | $k$   |       | 1     | $k$ |

(3.14)

The columns of the matrix are indexed by the events of the net. The entries in the columns indicate for each place the token increase or decrease upon the corresponding event occurrence. Given this matrix  $\underline{N}$ , the integer solutions  $x$  of  $\underline{N}^T \cdot x = 0$  are S-invariants. For the above example, two S-invariants,  $i_1$  and  $i_2$ , are presented in (3.14). Surprising or not, given an S-invariant  $i$  and an initial marking  $M_0$ , then for each reachable marking  $M$ ,

$$(3.15) \quad \sum_{s \in S} i(s) \cdot M(s) = \sum_{s \in S} i(s) \cdot M_0(s) .$$

Here,  $S$  is the set of all places of the net,  $i(s)$  is the  $s$ -indexed entry of  $i$ , and  $M(s)$  is the token count on the place  $s$  at the marking  $M$ . For the above example, we obtain with  $i=i_1$  the value  $n$  and with  $i=i_2$  the value  $k$ .

Further details on  $S$ -invariants as well as a proof of the equation (3.15) can be found in the literature, e.g. [Reisig 85a].

For our example follows from the invariant  $s_1$ , that the sum of tokens on the places  $s_0, \dots, s_4$ , i.e. the number of processes, remains constant. From  $i_2$  follows:  $s_4$  contains at each reachable marking at most one token; that is, there exists at most one writing process. When  $s_4$  carries a token then  $s_2$  and  $s_5$  are empty. So, while some process is writing, no other process reads the buffer. Furthermore follows that  $s_2$  carries at most  $k$  tokens: there are at most  $k$  processes reading concurrently. When no process is writing, that is,  $M(s_4)=0$ , then  $s_2$  may in fact obtain up to  $k$  tokens. Finally one can prove by help of the invariants, that in (3.13) no stillstand can occur, provided we start with  $n$  tokens at  $s_0$  and  $k$  tokens at  $s_5$ .

This should suffice to get a first idea about the principles and the power of  $S$ -invariants. For details we refer to [Reisig 85a]. A lot of work was done to extend this technique to models for nets with individual tokens. We refer to [Genrich 86], [Jensen 86] and [Reisig 85c] for details.

### 3.3 Further Analysis Techniques

We only mention some further analysis techniques in this section. On the base of  $S$ -invariants, where solutions of the equational system  $\underline{N}^T \cdot x = 0$  have been used to derive system properties, one may inquire solutions of  $N \cdot x = 0$  too. Such solutions in fact have a nice interpretation.  $x$  clearly assigns a value  $x(t)$  to each transition  $t$  of the net  $N$ . Assume all  $x(t)$  are natural numbers, and assume each event  $t$  to occur exactly  $x(t)$  times. By doing so, starting from an initial marking  $M$ , after the described event occurrences  $M$  is reached again.

A "classical" concern of Petri Net Theory is the investigation of the set of all reachable markings. This set is in general infinite, because the token count on places may grow unlimited. The "reachability problem" belongs to this area: given an arbitrarily chosen marking  $M$ , how to decide whether  $m$  is reachable or not? For more than a decade this was an open problem.

To systematically study the set of all reachable markings one may

construct the tree of reachable markings. Its nodes are labeled by markings and its arcs by events. The label of the root is the initial marking. Given a node labeled by a marking  $M$ , for each event  $t$  that may occur at  $M$ , there is an arc in the tree, labeled by  $t$ . The arc ends at a node which is labeled by the marking gained from  $t$ 's occurrence at  $M$ . This tree is in general infinite. On this base the finite "coverability tree" can be derived. Roughly, infinite paths with increasing markings as node labels, are replaced by single nodes. Some questions of liveness and boundedness can be attacked by the coverability tree.

Finally we mention the idea of net completion for the description and analysis of net properties. A given net may be augmented by an additional event  $t$  with the distinguished property that no reachable marking enables  $t$ . This may appear a useless construct, but the fact that  $t$  never occurs denotes an interesting property of the system. For certain classes of nets, such events correspond to certain classes of logically expressible system properties, called "facts".

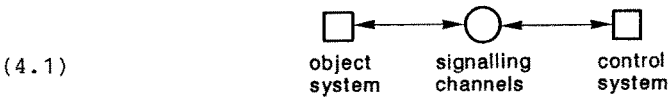
One likewise may augment places, and study the increase and decrease of their token load. This describes the degree of synchronization among the events that are directly linked to such places, thus motivating the notion "synchronic distance".

#### 4. SOME APPLICATION AREAS

##### 4.1 Process Control

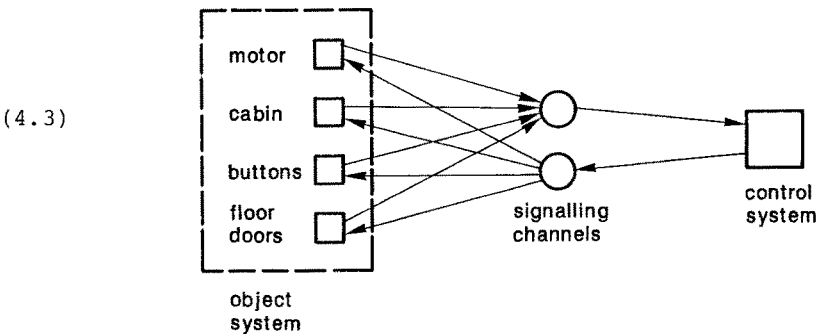
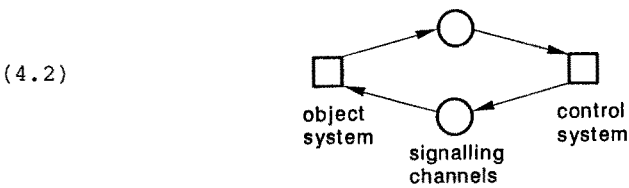
A typical example of an embedded system, taken from every-day life, is a lift control system. The difficulties of such a system does not consist in the complexity of the involved programs but the intricate interaction between the program and the lift cabine, the motor and the buttons pressed by customers. To cover all this, it is inevitable not only to model the lift control, but also the behaviour of the technical components of the lift.

We shall develop a lift model in the following. In this case and generally in case of embedded systems, it is useful to distinguish the object system, the control system, and the signalling channels that link both systems. (Figure 4.1)



In our lift model the object system includes the motor, the cabin, the customer's buttons, the floor doors, etc. The control system is to process signals to and from the components of the object system. These signals are transmitted via the signalling channel.

A nearby refinement of (4.1) is to distinguish the direction of signal flow, as shown in (4.2). A further refinement separates the components of the object system, and is shown in (4.3).



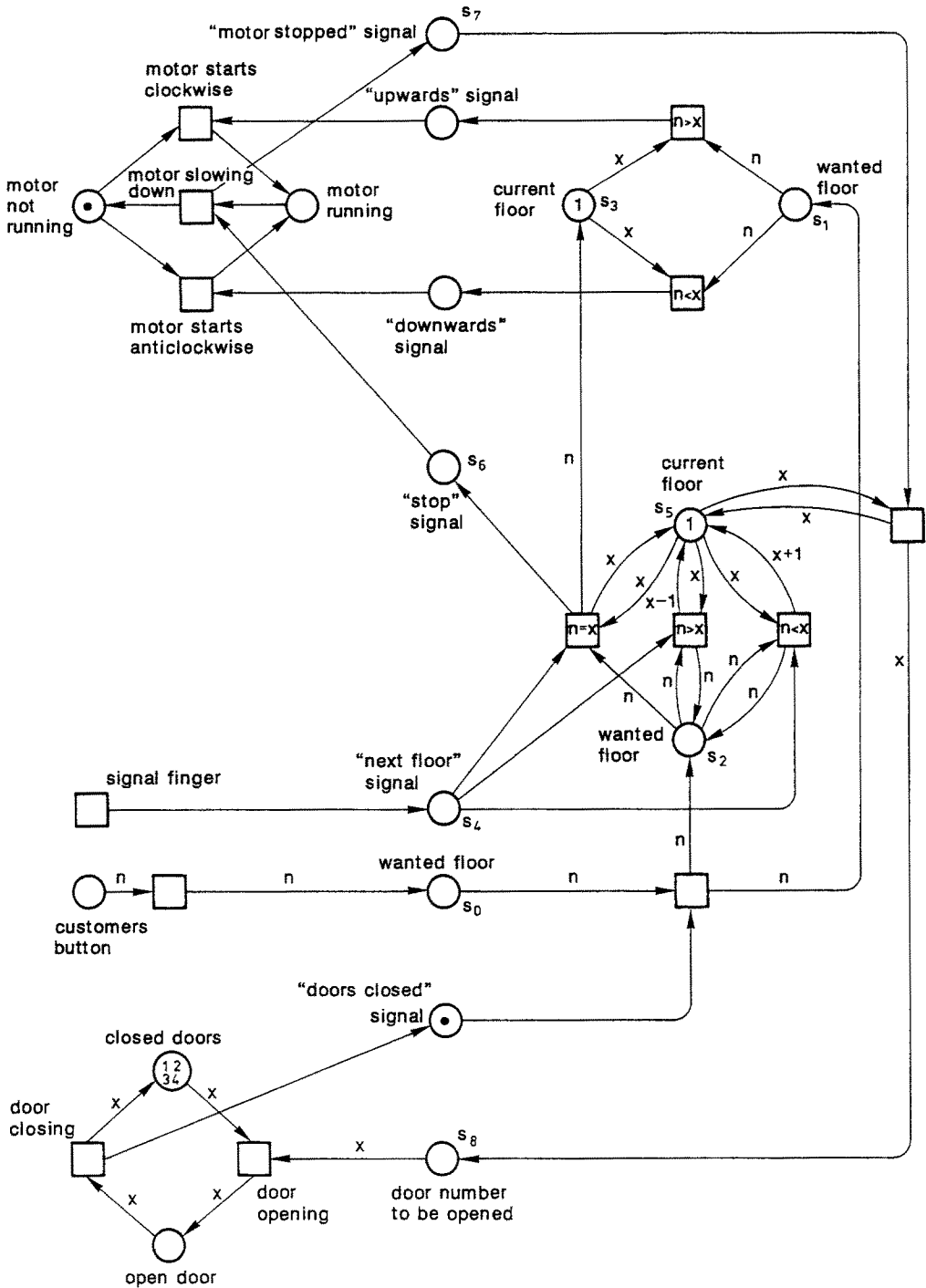
The next (and more or less final) refinement shows the overall behaviour of the system (Fig. 4.4). We assume a somewhat simplified behaviour of the lift system: The scheduling of concurrent requests is not included (though this would not be difficult), and upon serving a customer, no intermediate stops are possible. We use individual and "black" tokens in a mixed fashion. Some events are inscribed by additional constraints for their occurrence.

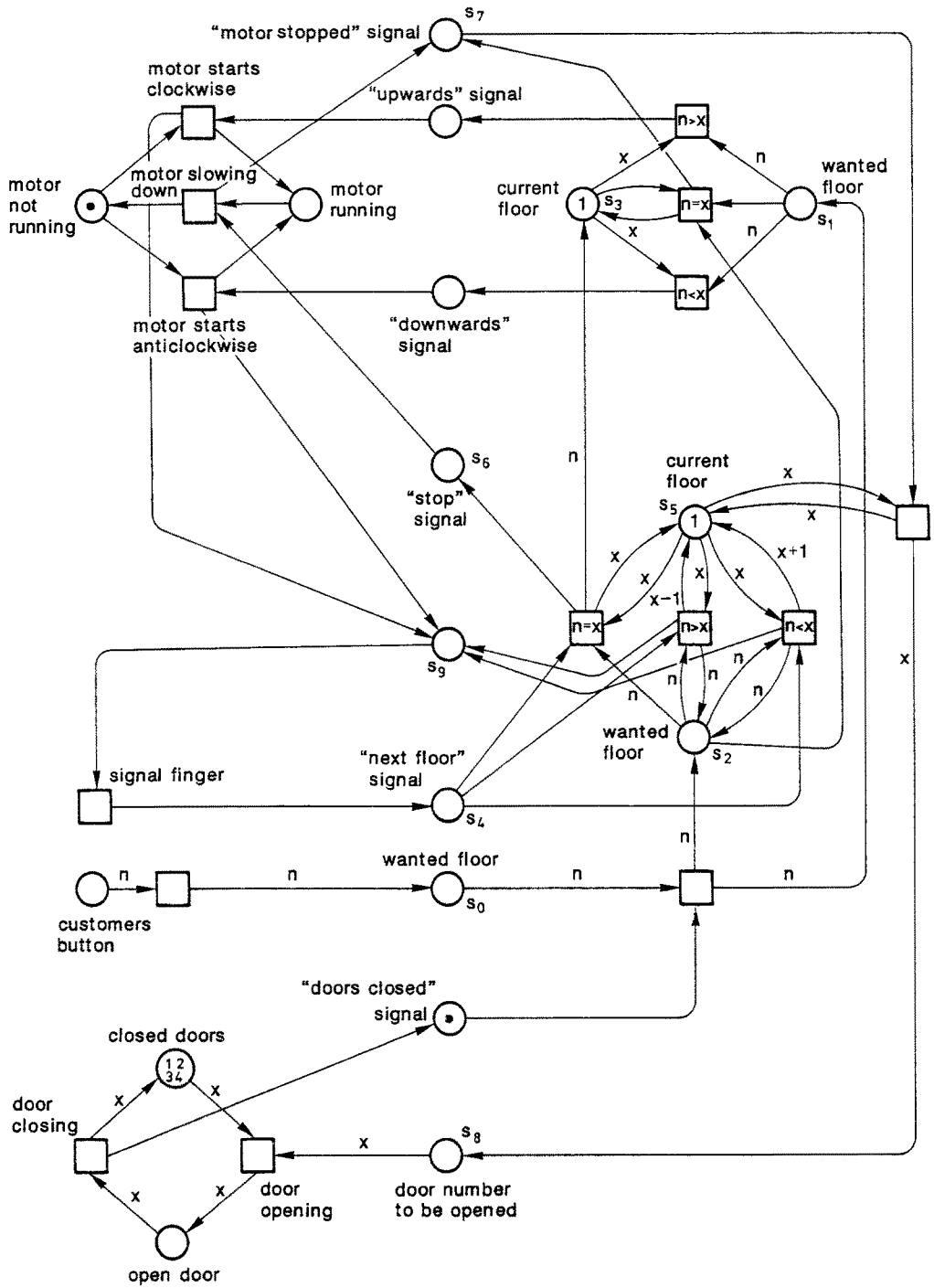
Let us follow a typical course of the lift. A customer may press a button and indicate the wanted floor. This results in an individual token,  $n$ , on place  $s_0$ . If the "door closed" signal is present, the information about the wanted floor,  $n$ , is copied into two places,  $s_1$  and  $s_2$ . The token in  $s_1$  is compared with the current floor, represented in  $s_3$ . (The meaning of the inscriptions of the involved events, " $x < n$ " and " $x > n$ ", respectively, should intuitively be clear. They are additional constraints for the events to occur). The outcome of these comparisons make the motor start either clockwise or anticlockwise.

We turn now to the signal finger. Between each two neighbour floors, a signal finger of the moving lift triggers a signal to a control unit (a black token on place  $s_4$ ). In this moment we defer the conditions to trigger this signal. The signal causes an update of the "current floor" token in place  $s_5$ . In case the current floor coincides with the wanted floor, place  $s_3$  is updated, too, and a stop signal is sent to the motor (place  $s_6$ ). So the motor slows down and sends a "motor stopped" signal to the control system ( $s_7$ ). Based on this signal, and on the number of the current floor in  $s_5$ , the number of the door to be opened is given to  $s_8$ . So the corresponding door will be opened; it will be closed after a while, and a "door closed" signal is available again, to serve the next customer.

A word on the signal finger is mandatory here. By construction of the lift object system it should be guaranteed that the lift triggers exactly one signal when moving in between two floors. We may model this constraint by an additional place  $s_9$  which in (4.5) is augmented to the net (4.4). With  $s_9$ , a signal can be given to  $s_4$  only after starting the motor (clockwise or anticlockwise), or in case the moving lift has not yet reached the wanted floor.

The system (4.4) sticks in case the floor wanted by the customer and the current floor of the lift coincide. In (4.5) this case is also considered: A "motor stopped" signal is triggered, causing the opening of the corresponding door.





The proper behaviour of the above lift system can now be proven by use of various techniques. S-invariants, as described in Section 3.2 for nets with "black" tokens, can be extended to nets with individual tokens. They can be used to show, for example, that never more than one door is open, that the motor is running only in case all doors are closed, etc. It is extremely useful for the deviation of system properties that the object- and the control-system are described in the same "language". The most important S-invariants coincidentally concern places of both systems as well as some signalling channels.

## 4.2 Data Flow Systems

Data Flow Systems are related to Petri nets with respect to the mechanisms by which computations are driven ahead. Without going into details of different such mechanisms, we discuss in the following a revised version of an example in [Kluge 85].

From a conceptual point of view, a computational process may simply be thought of as the evaluation of a function that is applied to a set of arguments. The result of the computation is a value. As an example, the function which computes the absolute value of two integers  $v$ , and  $u$ ,

$$(4.6) \quad |v-u| = \text{if } v > u \text{ then } v-u \text{ else } u-v$$

may be represented as



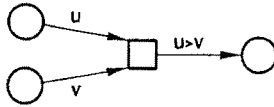
This operational unit may start operating whenever two values,  $u$  and  $v$ , are present in the corresponding places.

We may assume only quite elementary function units be available, namely difference



and comparison

(4.9)

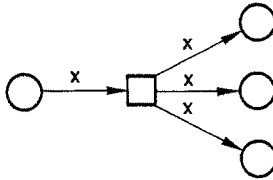


which returns the boolean value "true" in case  $u > v$ , and the value "false" in case  $v \geq u$ .

We furthermore assume universal functions, i.e. functions which can be applied in any context, independent of specific properties of objects and functions (such as subtraction).

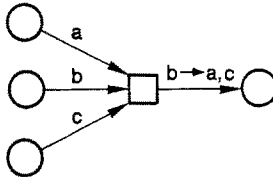
For our purposes we need the 3-copy function

(4.10)



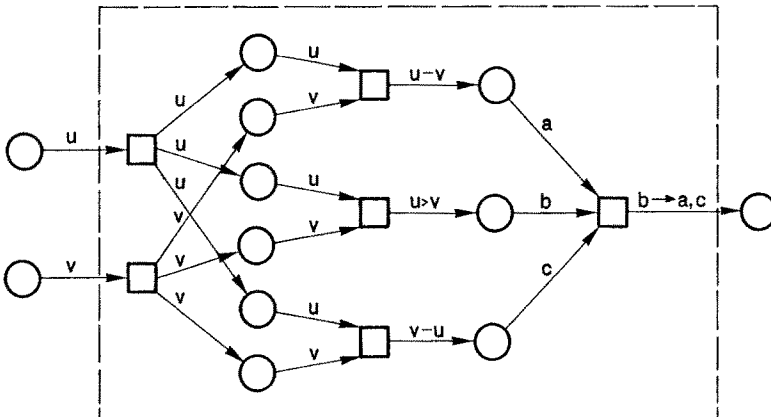
which returns three instances of its input object, and the conditional selection function

(4.11)



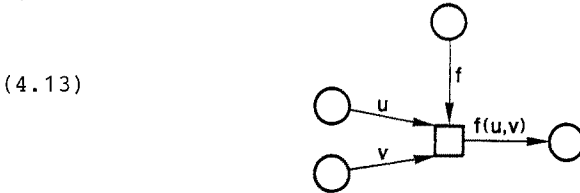
which expects a boolean  $b$  and transmits either of  $a$  and  $c$ , depending on the concrete value of  $b$ . The inscription  $b \rightarrow a, c$  reads "if  $b$  then  $a$ , else  $c$ ".

(4.12)

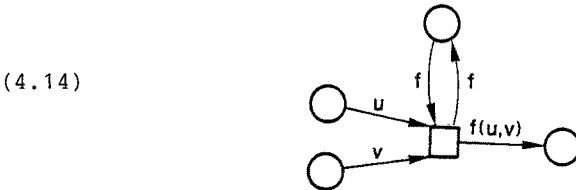


By help of the above functions we can now design a quick, parallel implementation of  $|u-v|$ , as a refinement of (4.7) to (4.12).

A more general principle of data flow architectures is to take functions as arguments. We then can do with operational primitives of the form



This means the same operational unit may be used to compute different functions. One may assume that the function to be applied is not "consumed" as the arguments to the functions are. So we end up with



For further details of such models of computational processes we refer to [Kluge 85]

### 4.3 Miscellaneous

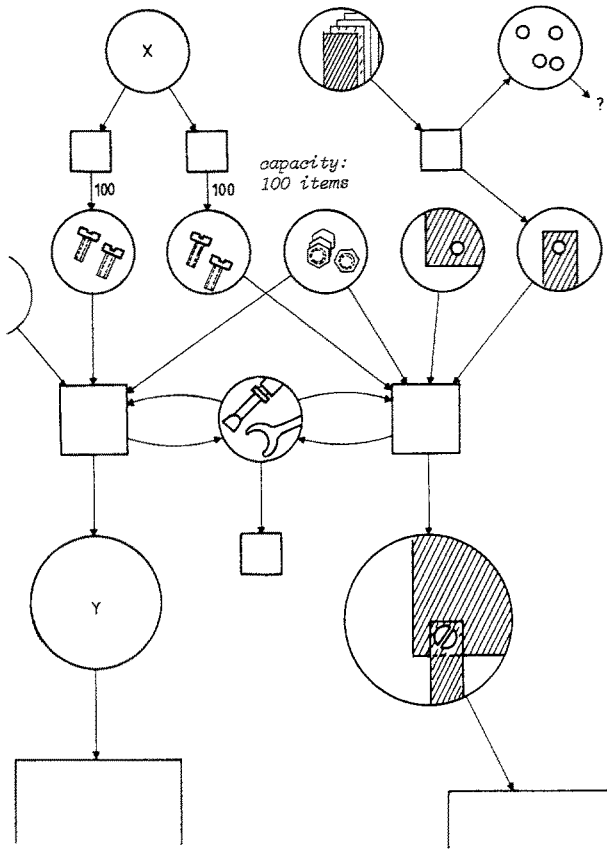
We just glance various application areas and examples for Petri net representations of embedded systems.

Figure (4.15) shows a fragment of an industrial production system, the operation of which is intuitively clear. Petri net models of manufacturing systems have been presented in [Kamath, Viswanadham 86], or [Murata et al 86].

An important area are office support systems, because the interface between the computer and its environment is the crucial problem (cf. Section 1.1). Quoting [Richter, Voss 86], Figure (4.16) shows a first step in modelling a passport authority.

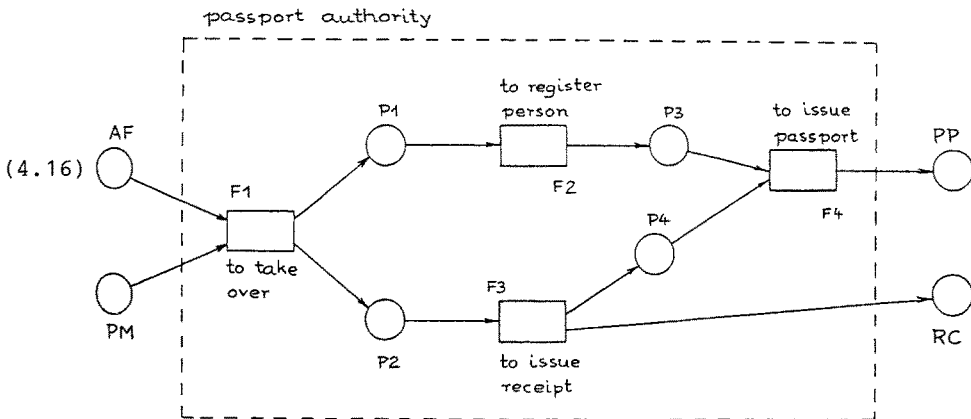
In a similar style, a simple part of an employment office which keeps track of employed and unemployed persons and of vacant and occupied positions, is in [Heuser, Richter 85] discussed as an example of the conceptual schema of a database.

(4.15)



Petri net based models of the interface between human and computers are discussed in [Oberquelle 84].

To come closer to computer organization, Petri Nets have successfully been applied for communication protocols. From the large number of papers in this area let us just mention [Diaz 82]. Problems of computer organization are modelled with Petri Nets in [Jessen, Valk 86]. Computerized process control is studied on the base of Petri Nets in [Schnieder 86].



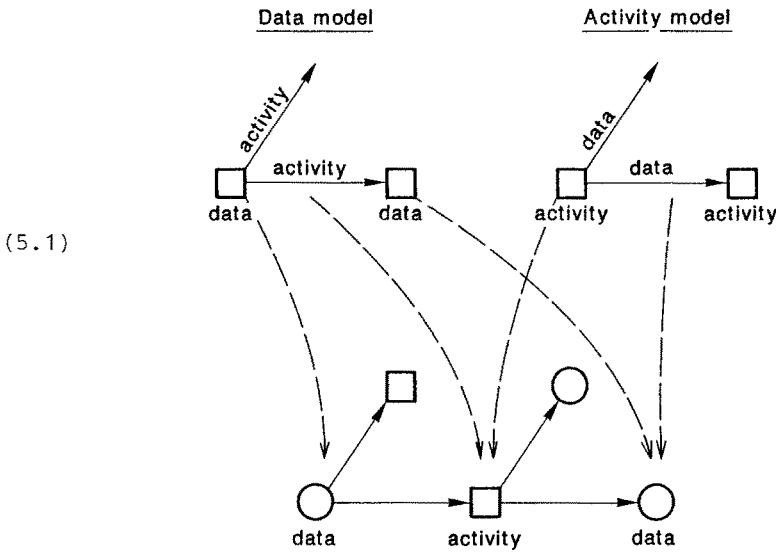
## 5. SOME RELATIONSHIPS TO OTHER METHODS

### 5.1 Petri Nets and SADT

There are a number of other established methods for "requirements engineering", in addition to nets. The "structured analysis and design technique" or SADT is a particularly well-known example [Ross, Schoman 77]. At first sight it appears similar to the procedure described in the first part of this article. SADT involves the drawing up of "activity diagrams" for the purpose of breaking down a system into a number of "activities". These activities are graphically represented as boxes and are connected with one another by means of "data arrows". Expressed in a somewhat abbreviated manner, these arrows interconnect "data producers" with "data consumers". "Data diagrams" are also drawn up, in which "data files" are depicted and interconnected by means of "activity arrows".

In the case of Petri nets, too, a careful distinction is made between the data and the activity aspects of system components. However, in contrast to SADT, they are modelled as circles or boxes on an equal level and simultaneously in the same graphic representation. The arrows in Petri net diagrams have a totally different function from those in SADT diagrams. In the section on "passive and active components" it was indicated that a Petri net arrow never represents a system component (i.e. an activity or a data file). Instead, it always represents an abstract relationship between such components. Figure 5.1 outlines these relationships.

Gradual, hierarchical refinement is common to virtually all software design methods, including Petri nets and SADT. In SADT, however, no further refinement is possible on moving



from merely breaking up the system down into single components to representations in which dynamic behaviour plays a role. It is impossible to formulate dynamic behavior in SADT (as well as in numerous similar methods). At best, dynamics can be indicated on the basis of carefully selected inscriptions.

In addition to the fact that nets provide an integrated representation of activity and data factors, they offer a further crucial

advantage, i.e. that of being able to represent dynamic behavior without having to leave the calculus. The formulation of dynamic behaviour is directly imposed on a net that only separates passive and active components. Examples for this step have been discussed above. We remind the step from (1.6) via (1.7) to (2.27), the step from (2.9) to (2.19), from (3.3) and (3.5) to (3.6), and from (4.2) to (4.4).

## 5.2 Petri nets and Jackson Structured Programming

To describe dynamic behaviour on the basis of a design formulated in SADT, it is necessary to switch to a different calculus. Jackson Structured Programming (JSP) is good in such cases. Other methods are also available such as "structograms" and "pseudocode".

JSP and "pseudocode" employ three "structuralizers" for the most part, i.e. "T;U" ("first T then U") for sequence formation, "T or U" ("either T or U") for either-or choices, and "T\*" ("T is repeated") for repetitions. The various graphic representations available for this purpose (tree structures in JSP, program language syntax in pseudocode) are not relevant in this context.

It doesn't take long to see that the three "structuralizers" described above can be formulated quite easily using Petri nets. What is involved for the most part is the representation of control flow, for which conditions/event systems are sufficient.

Nets have an additional structuralizer, in addition to the three already mentioned, i.e. "T and U" ("both T and U") for accumulations. It is always implicitly present in net graphs whenever different boxes are not directly connected to a circle. This structuralizer is extremely useful, since it can be used to provide a precise description of local transitions with spatially or logically limited causes and effects. This, in turn, makes it unnecessary to introduce "global states". In the case of large distributed systems, involving several active components, these states would not be intuitive anyway. This has already been discussed at various places above.

In our library example, books are requested and returned without the involvement of any temporal or causal order. The non-ordered occurrence of events is explicitly contained in net models. Sequences can, of course, be formulated wherever they are necessary or desired. By way of example, in Figure 11 the library can take back a book if and only if the book in question has previously been turned in at the return counter.

In general it can be said that sequence formation is an effective structuralizer only in those cases when it is also possible to do without it. Otherwise it is impossible to see whether a sequence results from a real system structure or whether it has only been formed because the available structuralizers made a given sequence necessary.

## CONCLUSION

We have outlined how Petri nets can be used to go from initial informal sketches to formal specifications of dynamic system behaviour without a need for switching representation models. Everything is developed on the basis of previous steps. There is a smooth transition from the definition of separate system components to the formulation of dynamic system behavior.

No other software engineering method covers as broad a spectrum of the software life cycle. SA and SADT only cover the initial analytic stages. JSD and "pseudocode" only cover the approximate formulation of dynamic behavior. Virtually none of the other methods in current use cover the role played by nets with individual tokens. Abstract data types might be considered as coming close to it. However, they don't have a concept for control and data flow. Nets with individual tokens are precisely the type of technique needed for "rapid prototyping".

Many principles for system design with nets are quite similar to techniques in current use, in particular the hierarchical separation of active and passive components as well as structuralizers for dynamic behavior. However, in every instance, net calculus goes beyond what other current methods are able to offer. In the case of requirements engineering it adds the integration of active and passive components in a single representation and the additional explicit representation of abstract relations between the components (arrows). In the case of dynamic behavior it adds the principle of local, independent transitions, i.e. "accumulation" as an additional structuralizer. These supplementary factors provide additional options for creating perspicuous and reality-based models. Above all, they have made it possible for the first time to integrate the entire design process in a single calculus.

Does this make Petri nets a complete system design method? In contrast to SADT, there is no indication of rules for arranging nodes, a primary direction for arrows (in SADT it is "from top left to bottom right"), or a guideline for the number of new nodes in a refinement step

(in SADT a maximum of seven). There is also a lack of conventions on the different factors involved and the roles that channels can play when agencies interact with them. The direction indicated by arrows reflects the input and output factor, but we have not undertaken a further breakdown into practical "aids" and "parameters". The same applies to conventions for graphic simplification and clarity (e.g. using one arrow in the place of several, organisation of design documentation etc.).

As important as they may be for practical procedure, these rules are not essential and not specific to SADT. Nothing stands in the way of their also being used for system design with nets, thus expanding the conceptual and graphic potential already contained in the net calculus for refinement or, in general, for the depiction of relationship between net representations. Thus, the principles of system design with nets described in this article can easily be supplemented to form a full-fledged method which exceeds the power of most known methods.

The graphic techniques for representing nets are simple enough to be easily accessible to the non-expert. As such, they are suitable as a means of communication between the various persons involved in a software project. Net models can consequently be used as a blueprint both for those who commission software development as well as for software engineers themselves. Similarly, they can also be used as a means of helping software users understand the programs in question.

This is different with the procedures required for net analysis, since a certain amount of mathematics is necessary to understand them. Still, the results of these analytic procedures can often be interpreted intuitively.

## REFERENCES

We mentioned only a small amount of work relevant to the subject of this paper. A comprehensive presentation of the state of the art in Petri nets is given at the "Advanced Course on Petri nets" in Bad Honnef Germany, 8.-19. September 1986. Its papers are published in [Brauer et al 87]. A lot of work on Petri nets is presented at the "European Workshops on Petri Nets" on a yearly basis. There are yearly volumes "Advances on Petri Nets", published by G. Rozenberg in the series Lecture Notes in Computer Science, of the Springer-Verlag. A bibliography on Petri Nets with 1800 entries, covering papers until spring 1986, is [Drees et al 86]. Recent references and an actual information service on

Petri Nets provides the "Petri Net Newsletter", publishing three issues per year. It is available at Gesellschaft für Informatik, Postfach 1669, 5300 Bonn, West Germany.

In the following list we shall not explicitly refer to papers of the above mentioned "Advanced Course on Petri Nets", and of the "Advances in Petri Nets" volumes.

[Apt 85] Krzysztof R. Apt: Logics and Models of Concurrent Systems. Springer-Verlag (1985)

[Brauer et al 87] W. Brauer, W. Reisig, G. Rozenberg: Advanced Course on Petri Nets. Lecture Notes in Computer Science, Springer-Verlag to appear 1987

[Diaz 82] Michel Diaz: Modeling and Analysis of Communication and Cooperation Protocols using Petri Net Based Models. Tutorial Paper, Proc. of the IFIP WG 6.1. Second Int. Workshop on Protocol Specification, Testing, and Verification, Idyllwild-CA, May 1982, C. Sunshine (ed.) North Holland (1982) also: Computer Networks, vol. 6 no 6 (1982)

[Dijkstra 71] Edsger W. Dijkstra: Hierarchical Ordering of Sequential Processes. Acta Informatica, 1, S. 115-138 (1971)

[Drees et al 86] Stefan Drees et al: Bibliography on Petri Nets. Gesellschaft für Mathematik und Datenverarbeitung (GMD), St. Augustin (Germany) internal report 222 (1986)

[Genrich 86] Hartmann J. Genrich: Net Theory and Application. Information Processing 86, H.J. Kugler (ed.), North Holland Publ. Comp., IFIP (1986) pp 823-831

[Heuser, Richter 85] Carlos A. Heuser, Gernot Richter: On the Relationship between Conceptual Schemata and Integrity Constraints on Databases. in: R. Meersman, T.B. Steel (eds). Proceedings of the IFIP Working Conference on Database Semantics, North Holland Publ. Comp., Amsterdam (1985)

[Jessen, Valk 86] Eike Jessen, Rüdiger Valk: Rechensysteme, Grundlagen der Modellbildung. Springer-Verlag (1986)

- [Kamath, Viswanadham 86] M. Kamath, N. Viswanadham: Application of Petri net based models in the modeling and analysis of flexible manufacturing systems. IEEE Conf. on Robotics and Automation, San Francisco p. 312-317 (1986)
- [Kluge, Lautenbach 82] Werner E. Kluge, Kurt Lautenbach: The Orderly Resolution of Memory Access Conflicts among Competing Channel Process. IEEE Transactions of Computers, Vol. C-31 pp 194-207 (1982)
- [Kluge 85] Werner Kluge: An Approach to Computer System Modeling Based on Petri Nets. in: Methodologies for Computer System Design, W.K. Giloi and B.D. Shriver (Editors) North-Holland Publ. Comp. IFIP, pp 143-160 (1985)
- [Murata et al 86] Tadao Murata et al: A Petri net based controller for flexible and maintainable sequence control and its applications in factory automation. IEEE transactions on industrial electronics, Vol. IE-33 no 1, Febr. 1986
- [Oberquellle 84] Horst Oberquellle: On Models and Modelling in Human-Computer Co-Operation. in: van der Veer et al (eds). Readings in Cognitive Ergonomics-Mind and Computers. LNCS 178 pp 26-43, Springer-Verlag (1984)
- [Petri 76] C.A. Petri: Interpretations of Net Theory. Gesellschaft für Mathematik und Datenverarbeitung, Internal Report 75-07, Second, revised edition 1976
- [Reisig 83] W. Reisig: System Design Using Petri Nets. Informatik-Fachberichte 74: Requirements Engineering. Hommel, Krönig (eds) Springer-Verlag, pp. 29-41 (1983)
- [Reisig 85a] Wolfgang Reisig: Petri Nets. Springer-Verlag (1985)
- [Reisig 85b] Wolfgang Reisig: Systementwurf mit Netzen. Springer-Verlag (1985) english translation in preparation.
- [Reisig 85c] Wolfgang Reisig: Petri Nets with Individual Tokens. Theoretical Computer Science 41, pp 185-213 (1985)
- [Richter,Voss 86] Gernot Richter, Klaus Voss: Towards a Comprehensive Office Model Integrating Information and Resources. in: [Rozenberg], Vol. 222

- [Richter 83] G. Richter: Netzmodelle für Bürokommunikation. Informatik-Spektrum, Nr. 6, S. 210-220 (1983) and Nr. 7, S. 28-40 (1984)
- [Rozenberg] G. Rozenberg: Advances in Petri Nets. Lecture Notes in Computer Science Vol. 188(1984), Vol. 222 (1985). (1986) to appear
- [Ross, Schoman 77] D.T. Ross, K.E. Schoman: Structured Analysis for Requirement Definition. IEEE-SE, Jan 1977
- [Schnieder 86] Eckehard Schnieder: Prozessinformatik. Vieweg-Verlag (Braunschweig) (1986)

# ELEMENTS FOR THE FORMAL DESCRIPTION OF SYSTEMS

Raymond T. Boute

Department of Computer Science

University of Nijmegen, The Netherlands

## ABSTRACT

This paper is an introductory overview of some key concepts that constitute a theoretical basis for the formal description of systems. Following a brief discussion of the criteria determining the usefulness of such concepts for system analysis and design, this overview addresses formal languages and automata, language semantics (with illustrations in trace theory and formal logic), abstract data types and functional formalisms. For what concerns the latter topic, special attention is given to the transformational style of reasoning and to the suitability for describing various kinds of systems. A more complete treatment of the functional approach and its applications is given in two companion papers.

## 0. INTRODUCTION

### 0.0 Formalisms versus formal systems

The terms "formalism" and "formal system" are treated in some contexts as synonyms, and in other contexts as quite disparate concepts. Hence, to avoid confusion, it is not superfluous to briefly explain the intermediate viewpoint taken throughout this paper.

a. A formal system consists of:

- A syntactic definition of a "formal language" [HARR78], which is a set of strings of symbols. Syntax is described by means of grammar rules constituting an unambiguous criterion for verifying whether or not a string of symbols is a "sentence" of the language, i.e. belongs to that set.
- A semantic definition, that is: an explicit assignment of meaning to the sentences of the language [STOY77]. Usually this is expressed by means of a meaning function mapping sentences to meanings.

The conjunction of syntactic and semantic definition gives rise to a set of formal (i.e. syntactic) manipulation rules for transforming sentences, called

conversion rules. A set of conversion rules is called consistent if all transformations described by these rules are meaning-preserving, and complete if all meaning-preserving transformations are described by these rules. A typical example of a formal system is formal logic [MEND79, LEVI74].

b. A formalism is a precise, complete notation in which symbols directly stand for their meaning. The emphasis is on ease and simplicity of expression and manipulation. Typical examples are algebra and calculus (in fact, most of mathematics).

Both description methods have their relative merits. Formal systems are necessary for the study of the meta-aspects of the language or notation. In some cases, the convention that meaning is assumed only when assigned explicitly also has a practical advantage, because it creates the possibility for dealing with different meanings in the same context without danger of confusion. An example in this respect is system semantics [BOUT84,86a]. When dealing with a single meaning only, formalisms provide a better balance between rigor, simplicity and intuition, as well as an easier linkage to other formalisms. An example is the Functional Temporal Calculus [BOUT85, 86b].

### 0.1 Selection criteria

In deciding which concepts to choose as the basis for formal system description, the practitioner and the theoretician may take somewhat different, albeit not necessarily conflicting, viewpoints. The practitioner looks for usefulness and ease of application, the theoretician for beauty and elegance. These properties have a tendency towards reinforcing each other, as is demonstrated in most traditional engineering disciplines. Perhaps the main reason why many results of theoretical computer science have little practical applicability is that they lack the elegance of concept and formulation that is the hallmark of good mathematics. Elaborating on the causes of this situation is beyond the scope of this paper. Lack of maturity of the discipline and excess of nomenclature and proof-theoretic technicalities are certainly important factors.

However, both viewpoints converge on the following criteria:

- a. Expressive power, i.e. the coverage provided in terms of:
  - Classes of systems that can be described

- . conceptual systems (abstract, mathematical structures)
- . real systems (concrete, physical structures) e.g. analog, digital, mechanical systems
- System aspects that can be expressed, e.g. structure, behaviour, performance
- Extensibility, i.e. linkage to other formalisms

Manipulative power, i.e. suitability in view of:

- Automatic interpretation (simulation, symbolic)
- Human interpretation (reasoning): analysis, design, transformation
- Balance between intuitive appeal and rigor (which reflects the maturity of the approach)

## 0.2 Rationale for this presentation

This paper is a thoroughly revised version of [BOUT80] with some changes in the selection of the material and in the organization.

- a. The choice of the material is intended to supply the preparatory information for the two companion papers [BOUT86a,b], yet is kept sufficiently general to serve as an introduction to the other papers in these proceedings as well.
- b. The organization of the material is not done by application area (problem-oriented) as is customary, but by conceptual approach (principle-oriented). As a consequence, some forewarning is necessary to avoid loss of coherence. By the same token, the examples play a twofold role: mainly to clarify the conceptual issues, and occasionally to show their relevance to particular application-oriented problems.
- c. The correspondence with other engineering disciplines is the basic paradigm throughout this paper. An illustration of the correspondence between electronics and the concepts introduced in this paper is given in table 0. The separation of the description in two parts, viz. a part dealing with objects and a part dealing with object compositions, will be a recurrent theme, even if not always elaborated explicitly. Therefore the reader may wish to refer to this table occasionally when studying the subsequent sections.

Table 0: Correspondence between formalisms in engineering and computer science.

| discipline                      | dealing<br>with objects                          | dealing with<br>object compositions                                                    |
|---------------------------------|--------------------------------------------------|----------------------------------------------------------------------------------------|
| electronics                     | device laws<br>$v = Ri$<br>$i = C \frac{dv}{dt}$ | Kirchhoff's laws<br>$\sum_{\text{loop}} v = 0$<br>$\sum_{\text{node}} i = 0$           |
| mathematical<br>logic           | axioms<br>$A \Rightarrow (B \Rightarrow A)$      | inference rules<br>$\frac{A, A \Rightarrow B}{B}$                                      |
| (imperative)<br>programming     | operations,<br>assignment<br>$a := b * c$        | control structures<br>. sequencing<br>. if...then...else<br>. while...do...            |
| (imperative)<br>program proving | axioms<br>$R(f/x)\{x := f\}R$                    | inference rules<br>$\frac{P\{Q\}R, R\{Q'\}S}{P\{Q;Q'\}S}$                              |
| functional<br>formalisms        | functions<br>$fx = \text{expression}$            | variables + abstraction<br>$f_0g = \lambda x.(f(gx))$<br>combinators<br>$f_0g = Bfg$   |
| functional<br>proofs            | equalities<br>$x(y+z)=xy+xz$                     | substitution<br>$\frac{fx=\text{expr}, g(x,y) = h(fx,dy)}{g(x,y) = h(\text{expr},dy)}$ |

### 0.3 Notational conventions

We write  $x:X$  to indicate that  $x$  is an element of the set  $X$  or, conversely, that  $X$  is the type of  $x$ . Furthermore,  $X \rightarrow Y$  designates the set of all functions from  $X$  to  $Y$ . Hence  $f : X \rightarrow Y$  means that (the function)  $f$  associates with every element  $x$  in  $X$  a unique element in  $Y$ . This element is designated by  $fx$  (parentheses and spaces being used only for layout and emphasis or to overrule the usual precedence conventions).

We always carefully distinguish between a function  $f$  and its application  $fx$  to an argument  $x$ , because these are different mathematical objects, e.g. if  $f : X \rightarrow Y$  and  $x:X$  then  $fx : Y$ . Common practice in calculus and analysis is somewhat sloppy in this respect: one often finds sentences such as "let  $f(x)$  be a function of a real variable  $x$ " meaning "let  $f$  be a function over the real numbers".

Functions may also take functions as argument and yield functions as results. For instance,  $f : X \rightarrow (Y \rightarrow Z)$  means that  $f$  is a function such that, for every  $x$  in  $X$ ,  $fx$  is a function from  $Y$  to  $Z$ . This means in turn that, for every  $y$  in  $Y$ ,  $(fx)y$  is a value in  $Z$ . To reduce the number of parentheses, the common convention is that  $fx$  stands for  $(fx)y$  and  $X \rightarrow Y \rightarrow Z$  stands for  $X \rightarrow (Y \rightarrow Z)$ . Function composition is denoted by  $\circ$ , that is to say:  $(f \circ g)x$  is defined as  $f(gx)$ .

The primitive logical connectives are written  $\wedge$  (for "and"),  $\vee$  (for "or"),  $\sim$  (for "not"). For the derived logical connectives we write  $\Rightarrow$  (for "implies":  $a \Rightarrow b$  means  $\sim a \vee b$ ) and  $\Leftrightarrow$  (for "is logically equivalent to":  $a \Leftrightarrow b$  means  $(a \Rightarrow b) \wedge (b \Rightarrow a)$ ).

For quantification we use the following conventions:

Ax.P means: for all  $x$ , the property  $P$  holds

Ex.P means: for some  $x$ , the property  $P$  holds

and the abbreviations

Ax:X.P, meaning: for all  $x$  in  $X$ ,  $P$  holds

(abbreviation of Ax.( $x:X \Rightarrow P$ ))

Ex:X.P, meaning: for some  $x$  in  $X$ ,  $P$  holds

(abbreviation of Ex.( $x:X \wedge P$ ))

Notice that, from the correspondence Ax.P  $\Leftrightarrow$   $\sim$ Ex. $\sim$ P it follows that Ax:X.P  $\Leftrightarrow$   $\sim$ Ex:X. $\sim$ P.

Finally, for conditional expression we write " $c \rightarrow a ; b$ " as an abbreviation for "if  $c$  then  $a$  else  $b$ ".

## 1. OVERVIEW OF THE KEY CONCEPTS.

### 1.0 Relationship between the concepts.

For the sake of coherence, we briefly elaborate on the interrelation between the various concepts discussed.

#### a. General aspects of formal languages and automata

As mentioned in the introduction (subsection 0.0.a), a formal language is characterized by two elements:

- form (syntax): definition of sentences
- meaning (semantics): interpretation of sentences

An automaton is a (real or conceptual) mechanism for generating or processing sentences. Hence automata theory and language (syntax) theory are two approaches to the same subject; which of the two formulations is more appropriate depends on the problem at hand.

b. Specific language issues discussed in this paper:

- Usage of a meta-language to describe other languages (grammars to describe syntax, meaning functions to describe semantics)
- Example: a language for describing device interaction (concurrency)
- Example: the language of (propositional) logic
- "Abstract" data type definitions (axiomatic approach) and the relation to languages
- Functional formalisms (constructive approach)

We hope to convince the reader that the simple and intuitive concept of "function" is sufficiently "abstract" for all useful purposes (as has always been the case in most of applied mathematics).

### 1.1 Formal languages and automata [HARR78].

a. Definitions

- A language  $L$  over an alphabet (set of symbols)  $A$  is a set of finite strings composed of symbols in  $A$ . It is customary to write  $\epsilon$  for the empty string and  $A^*$  for the set of all strings (including  $\epsilon$ ) over  $A$ .

Examples of languages over  $A = \{a, b, c\}$  are:

$$L_0 = A^* \quad L_1 = \{\epsilon, a, cb\} \quad L_2 = \{a^n b^n \mid n: \mathbb{N}\}$$

where  $a^n$  denotes a string consisting of  $n$   $a$ 's. Notice that  $L_1$  is finite,  $L_0$  and  $L_2$  infinite. In general, since  $L$  is always a subset of  $A^*$ ,  $L$  is countable if  $A$  is finite (sufficient condition).

- A grammar is a precise set of rules for specifying a language. There are two styles of definition:
  - . using natural language (e.g. in most books on mathematical logic)
  - . using a formal grammar (e.g. for most programming languages).

The most common way to define a formal grammar is by means of a set of production rules describing a constructive way to generate all sentences of the language being defined. The usage of production rules is illustrated by the following example.

Example:     $START \rightarrow \underline{a}Ab$                     (0)  
               $\underline{a}A \rightarrow \underline{aa}Ab$                     (1)  
               $A \rightarrow \epsilon$                             (2)

The underlined symbols (called "terminal symbols") constitute the alphabet. Non-underlined letters or words are auxiliary symbols ("nonterminal symbols") whose role will become clear immediately. Other non-underlined symbols (in this case  $\rightarrow$ ) belong to the grammar notation. The production rules are used as follows: given a string consisting of terminal and/or nonterminal symbols, select any production rule whose left-hand side matches part of that string, and replace that part by the right-hand side.

Example: given the string  $abAbaAb$ , either occurrence of  $A$  may be replaced by  $\epsilon$  according to rule (2), and the (single) occurrence of  $aA$  may be replaced by  $aaAb$  according to rule (1).

The language is then defined as the set of all strings consisting of terminal symbols only that can be generated from  $START$  in a finite number of steps, each step using one of the production rules. For our example:

$START \rightarrow aAb \rightarrow aaAbb \rightarrow aaaAbbb \rightarrow aaabbb$   
           (0)    (1)        (1)        (2)

It is easy to see that the language defined by the set of production rules (0..2) is  $L_2 - \{\epsilon\}$ .

#### b. Discussion: Chomsky classification of languages.

By imposing restrictions on the kind of production rules being allowed, Chomsky introduces the following language classification:

- Class 0 - unrestricted grammars (no restriction).
- Class 1 - context-sensitive grammars (CSGs).  
     Restriction: the number of symbols on the left-hand side of  $\rightarrow$  must not exceed the number of symbols on the right-hand side.
- Class 2 - context-free grammars (CFGs).  
     Restriction: the left-hand side of  $\rightarrow$  must be a single nonterminal.
- Class 3 - linear grammars.  
     Restriction: context-free grammar where the right-hand side of  $\rightarrow$  is a string of terminal symbols followed by at most one nonterminal.

Clearly class 3  $<$  class 2  $<$  class 1  $<$  class 0. In our example, the production rule (1) suggests class 1, but it is easy to see that, in conjunction with (0) and (2), it may be replaced by  $A \rightarrow \underline{a}Ab$ . Hence the language  $L_2 - \{\epsilon\}$  (and  $L_2$  itself) can be described by a context-free grammar.

For what concerns practical significance, we note that class 3 is used in conjunction with finite-state machines (an important model for hardware design), whereas class 2 is used for the definition of programming languages, even though realistic programming languages are in fact context-sensitive and require augmenting a CFG by separate restrictions. The reason for not directly using CSGs is the relative unreadability of class 1 (and class 0) grammars. However, improvements are possible due to the introduction of two-level or van Wijngaarden grammars [CLEA77] and recent work on

- Affix grammars [KOST71] and extensions thereof [MEIJ86]
- Term Rewrite Systems or TRSs [KLOP85]

Another reason why Term Rewrite Systems are of interest to us is that they offer considerable promise as a universal basis for functional languages [EEKE86], viz.

- for describing reduction strategies,
- as a vehicle for implementation,
- for semantic definition,
- as an actual programming or description language.

#### c. Relation to automata

In general, an automaton is a real or conceptual mechanism that realizes a function  $f : L \rightarrow L'$  mapping sentences of a language  $L$  onto sentences of a language  $L'$ . An important kind of automaton is an acceptor for a given language  $L$  (over an alphabet  $A$ ), specified as follows

$$\text{acc}_L : A^* \rightarrow \{0,1\}^* \text{ (type definition)}$$

with, for the empty string,

$$\text{acc}_L \epsilon = \epsilon : L \rightarrow 1 ; 0$$

and, for any  $x$  in  $A^*$  and any  $a$  in  $A$  (i.e. for nonempty strings),

$$\text{acc}_L (xa) = yb \text{ where } y = \text{acc}_L x$$

$$b = xa : L \rightarrow 1 ; 0$$

Example:  $\text{acc}_{L_2} aaabbbb = 10000010$

Remark: the uninitiated reader as well as the overly meticulous one may be confused by the double usage of the symbols  $a$  and  $b$ . In the definition of the function  $\text{acc}_L$ ,  $a$  and  $b$  are "syntactic" variables designating arbitrary elements of  $A$  and  $\{0,1\}$  respectively. In the example,  $a$  and  $b$  are constants standing for themselves as specific elements of the alphabet  $A = \{a, b, c\}$ .

(end of remark)

The Chomsky language classes correspond to classes of automata in the following way:

- class 0  $\leftrightarrow$  Turing machines
- class 1  $\leftrightarrow$  Linear bounded automata
- class 2  $\leftrightarrow$  Pushdown automata
- class 3  $\leftrightarrow$  Finite-state automata

For the elaboration of this correspondence, we refer to [HARR78]

#### d. Simplified BNF ("Backus Normal Form" or "Backus-Naur Form")

In the sequel of this paper, we will describe context-free grammars by means of a simplified version of BNF inspired by Wirth [WIRT77]. Our notational conventions for the production rules are:

- we write = instead of  $\rightarrow$
- production rules with the same left-hand side, e.g.

A = alternative 0

...

A = alternative k

are grouped as follows

A = alternative 0 | ... | alternative k

For instance,  $L_2$  is described by  $START = \epsilon \mid \underline{a} \, START \, \underline{b}$ .

- Triples of points in a string of symbols denote ellipsis; e.g.

(expr, ... , expr)

denotes a pair of parantheses surrounding zero or more "expr"s separated by commas.

- Text between square brackets [] is meant as comment.

Context-sensitive grammars are described by BNF rules together with a separate set of restrictions.

## 1.2 Language semantics [STOY77]

### a. Definitions

Let L be a language (set of sentences) and D a set of meanings we wish to assign to sentences. Then a meaning function or semantic function is a function

$m : L \rightarrow D$

such that, for any sentence s in L,  $m_s$  is the meaning of s. The set D is called the domain of interpretation. Together with the meaning function, it constitutes a model for the language, e.g.  $M = \langle D, m \rangle$ .

Usually the definition of  $m$  is built hierarchically from auxiliary functions, one for each nonterminal symbol. The domain of such an auxiliary function is the set of all strings generated from that nonterminal symbol by means of the production rules. Such a set is called a syntactic category and the meanings assigned to the strings of such a set constitute a semantic category. An extensive example will follow soon.

#### b. Syntax, semantics and formal manipulation

Formal manipulation rules are rules for manipulating sentences on the basis of purely syntactic criteria, yet guaranteeing that thereby some semantic criterion is met. Examples are

- Conversion rules: for transforming sentences into other sentences that have the same meaning. These rules form the basis of transformational reasoning in functional formalisms (see section 2).
- Deduction rules: for combining sentences that represent true statements in order to obtain new true statements. These rules form the basis of deductive reasoning in logic (see section 1.4).

### 1.3 Example: a language for describing trace structures

#### a. Motivation

This example is chosen because it not only provides a good illustration of language syntax and semantics, but also because of the usefulness of trace structures in the following respects:

- For describing interaction between devices (with applications to VLSI design) [REMM85],
- As an abstract model for describing concurrency [KALD86].

Trace theory constitutes an interesting formalism for describing the behaviour of compositions of objects.

#### b. A bit of trace theory

A trace structure is a pair  $\langle A, X \rangle$  where  $A$  is an alphabet and  $X$  (the trace set) is a subset of  $A^*$ . As a matter of notation, if  $T = \langle A, X \rangle$  is a trace structure, then  $\underline{a}T = A$  and  $\underline{t}T = X$ .

A trace structure can be used to describe the behaviour of a system or component by means of sequences of symbols representing interactions with the environment or with other components.

Let  $B$  be an alphabet. Then we define the projection operation  $\upharpoonright$  as follows:

. for traces (i.e. elements of  $\underline{tT}$ ),  $\upharpoonright$  is formally defined by:

$$\epsilon \upharpoonright B = \epsilon \quad \text{and, for any } s \text{ in } A^* \text{ and } a \text{ in } A,$$

$$sa \upharpoonright B = a:B \rightarrow (s \upharpoonright B)a ; s \upharpoonright B$$

Thus projection on  $B$  retains only symbols belonging to  $B$ .

. for trace structures:

$$T \upharpoonright B = \langle \underline{aT} \cap B, \{s \upharpoonright B \mid s: \underline{tT}\} \rangle$$

The interaction between components can be described as follows. Assuming that common symbols in the alphabets represent mutual synchronization (this is only a matter of the proper choice of symbols), then the interaction between components is reflected by the weaving operation  $\underline{w}$ . For trace structures  $S$  and  $T$ ,

$$S \underline{w} T = \langle A, \{v:A^* \mid (v \upharpoonright \underline{aS}): \underline{tS} \wedge (v \upharpoonright \underline{aT}): \underline{tT}\} \rangle \quad \text{where } A = \underline{aS} \cup \underline{aT}$$

Example: if  $S = \langle \{a, b\}, \{\epsilon, a, ab, aba, abab, \dots\} \rangle$

and  $T = \langle \{b, c\}, \{\epsilon, b, bc, bcb, bc bc, \dots\} \rangle$

represent component behaviour characterized by alternating  $(a, b)$  and  $(b, c)$  respectively, then  $S \underline{w} T = \langle \{a, b, c\}, \{\epsilon, a, ab, aba, abc, abac, \dots\} \rangle$

(end of example)

Notice that  $\underline{w}$  is associative and commutative. A nontrivial theorem is the distributivity of  $\upharpoonright$  over  $\underline{w}$ :

Theorem: if  $\underline{aS} \cap \underline{aT} \subseteq A$  then  $(S \underline{w} T) \upharpoonright A = (S \upharpoonright A) \underline{w} (T \upharpoonright A)$

Hiding the internal synchronization (keeping only the symbols representing external interaction) is reflected by the blending operation  $\underline{b}$  defined by

$$S \underline{b} T = (S \underline{w} T) \upharpoonright (\underline{aS} + \underline{aT})$$

where  $+$  denotes the symmetric difference of two sets ( $A + B = (A \cap \bar{B}) \cup (\bar{A} \cap B)$ ) or, equivalently,  $(A \cup B) - (A \cap B)$ . In general,  $\underline{b}$  is not associative.

Example: using the same examples for  $S$  and  $T$  we obtain:

$$S \underline{b} T = \langle \{a, c\}, \{\epsilon, a, aa, ac, aac, \dots\} \rangle$$

(end of example)

The trace structures used in the example can be generalized to the concept of a k-level semaphore, defined as follows:

$$SEM_k(a, b) = \langle \{a, b\}, \{t: \{a, b\}^* \mid \underline{As} \leq t. (0 \leq l(s \upharpoonright \{a\}) - l(s \upharpoonright \{b\}) \leq k)\} \rangle$$

(Notation:  $s \leq t$  indicates that  $s$  is a prefix of  $t$ , and  $l s$  is the length of string  $s$ )

Theorem:  $SEM_m(a, b) \subseteq SEM_n(b, c) = SEM_{m+n}(a, c)$

### c. Language syntax

The syntax is defined by the following context-free part:

```

process = com alphabet ; command moc
  alphabet   = [set of symbols in set notation { ... }]
  command    =  $\epsilon$  | symbol | command*
              | (command ; command)
              | (command | command)
              | (command , command)

```

together with the restrictions defined under d.

### d. Language semantics

Let P be the syntactic domain of "processes" and C the syntactic domain of "commands". The domain of interpretation D being the set of trace structures, we define the semantic functions

$mproc : P \rightarrow D$  and  $mcom : C \rightarrow D$ ,

in the following way:

```

mcom  $\epsilon$  =  $\langle \emptyset, \{\epsilon\} \rangle$ 
mcom a =  $\langle \{a\}, \{a\} \rangle$  for a "symbol" a
mcom  $[[c^*]]$  =  $\langle \underline{a}(mcom\ c), (\underline{t}(mcom\ c))^* \rangle$  for "command" c
mcom  $[[c;c']]]$  =  $\langle \underline{a}(mcom\ c) \cup \underline{a}(mcom\ c'), \{uv \mid u:\underline{t}(mcom\ c), v:\underline{t}(mcom\ c')\} \rangle$ 
mcom  $[[c|c']]]$  =  $\langle \underline{a}(mcom\ c) \cup \underline{a}(mcom\ c'), \underline{t}(mcom\ c) \cup \underline{t}(mcom\ c') \rangle$ 
mcom  $[[c,c']]]$  =  $(mcom\ c) \underline{w} (mcom\ c')$ 
mproc  $[[com\ A : c\ moc]]$  =  $pref\ (mcom\ c)$ 

```

where  $pref : D \rightarrow D$  ("prefix closure") is defined by

$pref\ T = \langle \underline{a}T, \{u \mid \underline{Ev}:\underline{t}T.u \leq v\} \rangle$

The double square brackets are just quote marks for "long" syntactic expressions.

The syntactic restrictions imposed on a process of the form

$com\ A : c\ moc$

are defined using the semantic functions as an intermediary, viz.

$\underline{a}(mcom\ c) = A$  and  $\underline{t}(mcom\ c) \neq \emptyset$

### e. Illustration

- Example of a "legal" process describing a 1-level semaphore:

$sem_1(a, b) = com\ \{a, b\} : (a;b)^* moc$

Notice that;

$$\text{mcom } \llbracket (a;b)^* \rrbracket = \langle \{a, b\}, \{\epsilon, ab, abab, \dots\} \rangle$$

$$\text{mproc } \llbracket \text{com } \{a, b\} : (a;b)^* \text{ moc} \rrbracket \langle \{a, b\}, \{\epsilon, a, ab, aba, \dots\} \rangle$$

- Example of an "illegal" process:

$$\text{ill} = \text{com } \{a\} : (a, (a;a)) \text{ moc}$$

Notice that

$$\begin{aligned} \text{mcom } \llbracket (a, (a;a)) \rrbracket &= \langle \{a\}, \{t \mid t \upharpoonright \{a\} = a \wedge t \upharpoonright^+ \{a\} = aa\} \rangle \\ &= \langle \{a\}, \emptyset \rangle \end{aligned}$$

#### f. Related concepts

For the sake of completeness, we mention the following formalisms related to trace theory.

- Milner's Calculus of Communicating Systems (CCS) [MILN80]

The main difference with trace theory is that, of the two distributivity laws in trace theory

$$(a|b);c = (a;c)|(b;c)$$

$$a;(b|c) = (a;b)|(a;c)$$

only the first one appears as an axiom in CCS. This different view in CCS is meant to reflect that  $a;(b|c)$  means: after the occurrence of event  $a$ , both  $b$  and  $c$  are still possible, whereas  $(a;b)|(a;c)$  indicates that the occurrence of event  $a$  implies that an alternative has been chosen already, and hence after  $a$  only either  $b$  or  $c$  is possible (although it is not known which one).

- Bergstra and Klop's Process Algebra [BERG85]

This is a more elegant formulation (extended with many variants) of CCS.

The problems with this class of formalisms are

- The existence of too many variants with insufficiently significant differences
- The fact that they yield very cumbersome expressions (unwieldy for human interpretation) even for simple problems, with little hope for simplification because there appear to be no truly "powerful" theorems thus far.

### 1.4 The language of logic [MEND79]

#### a. Syntactic aspects of propositional logic (PL)

The language syntax for PF is:  $W = V \mid \neg W \mid (W \Rightarrow W)$

$$V = \underline{a} \mid \underline{b} \mid \underline{c} \mid \dots$$

introducing the syntactic categories  $W$  (well-formed formulas or wffs) and  $V$  (variables).

The formal manipulation rules or "deduction rules" are:

- the logic axioms, i.e. the following predefined set of wffs:

$$A \Rightarrow (B \Rightarrow A) \quad (\alpha)$$

$$(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C)) \quad (\beta)$$

$$(\neg B \Rightarrow \neg A) \Rightarrow ((\neg B \Rightarrow A) \Rightarrow B) \quad (\gamma)$$

where A, B, C are any wffs

- a single rule of inference (for "combining" axioms):

$$\frac{A, A \Rightarrow B}{B} \quad \text{called } \underline{\text{modus ponens}} \quad (\text{MP})$$

The following definitions indicate the usage of the deduction rules:

- A theory is a predefined set of wffs.
- A proof on a theory T is a sequence of wffs, each of which is either (a) a logic axiom, (b) a wff in T, (c) the result of applying MP to wffs occurring earlier in the sequence.
- A theorem of T is the last wff of a proof on T

Notation:  $T \vdash W$  means that W is a theorem of T.

Example: let  $T = \{a\}$

Theorem:  $b \Rightarrow a$

Proof: 0. a (since a is a wff in T)

1.  $(a \Rightarrow (b \Rightarrow a))$  (logic axiom  $\alpha$ )

2.  $b \Rightarrow a$  (MP applied to 0. and 1.)

Hence the metatheorem  $\{a\} \vdash b \Rightarrow a$ , asserting the existence of a proof for  $b \Rightarrow a$  on the theory  $\{a\}$ . (end of example)

Metatheorem (Herbrand's theorem)

If  $T \cup \{A\} \vdash B$  then  $T \vdash (A \Rightarrow B)$  (end of metatheorem)

Herbrand's theorem reflects an often-used strategy in theorem proving: to prove that  $A \Rightarrow B$ , assume A (as an axiom) and deduce B.

## b. Semantics of propositional logic

The domain of interpretation  $D = \{0, 1\}$  where 0 stands for "false" and 1 for "true". For obvious reasons, the meaning function must be parameterized by an interpretation for the variables (because the meaning of a wff depends on the meaning of the variables in that wff). If we let  $I = V \rightarrow D$ , then the meaning function for wffs is defined by

$m : W \rightarrow I \rightarrow D$  with

$m \ v \ i = i \ v$  (for any variable v in W)

$$m \llbracket \neg w \rrbracket i = (m w i) = 1 \rightarrow 0 ; 1 \text{ (for any wff } w \text{ in } W)$$

$$m \llbracket w \Rightarrow w' \rrbracket i = (m w i) = 1 \rightarrow m w' i ; 1 \text{ (for any wffs } w \text{ and } w' \text{ in } W)$$

Readers who are puzzled by this definition should refer back to subsection 0.3 or forward to 2.1.

We now define the concept of semantic implication, written  $\models$ . For a theory  $T$  and wff  $X$ , the notation  $T \models X$  means:

if  $m Y i$  for all  $Y$  in  $T$  and all  $i$  in  $I$ , then  $m X i$  for all  $i$  in  $I$ .

Deduction (a syntactic operation) can now be related to semantics by means of the following metatheorem.

Metatheorem: the deduction rules of PL are complete and consistent, i.e.

- if  $T \vdash X$  then  $T \models X$  (consistency)
- if  $T \models X$  then  $T \vdash X$  (completeness) (end of theorem)

In words: every true formula has a proof, and every formula that has a proof is true (all w.r.t. theory  $T$ ).

#### c. Concluding remarks

The language PL has been introduced for three reasons

- to illustrate language syntax and semantics,
- as background material for the abstract data type example in the Appendix,
- as a prologomenon to the discussion on system semantics [BOUT86a].

For the extension (or rather: refinement) of PL to the predicate calculus through the addition of functions, predicates and quantifiers, we refer to the relevant literature [MEND79, LEVI74].

### 1.5 Abstract data types

#### a. Purpose

Abstract data types are intended to describe values (in the most general sense), and operations on these values. They are abstract in the sense that values are defined by the effect that operations have on them, not by some representation. Abstraction is meant to exclude irrelevant information from a description.

In the context of this course on embedded systems, ADTs are also the starting point for defining the "concrete data types" by which they are realized, for instance the behaviour digital systems is expressed in terms of concrete data types.

## b. Style of definition

ADT definitions usually consist of two parts:

- A type definition for the values and the operations. This definition specifies which combinations are "legal", and hence is of a syntactic nature (in a sense that will be made more precise later on). On this point, there are few differences between the various styles.
- A definition of the effect of the operations (in principle a semantic issue). Here one finds various styles:
  - . Informal (natural language).
  - . Operational (or procedural), as in most programming languages with "module" facilities. This description style originated with the programming language SIMULA.
  - . Equational (or nonprocedural), further subdivided in:
    - algebraic/axiomatic style [STAN78];
    - algebraic/categorical style [GOGU78];
    - algebraic/constructive style [TURN85].
  - . Pre- and postconditions (logic style) [WULF76].
 and so on.

## c. An example; a "bag of elements".

Informally, a "bag" is similar to a set, except for the fact that it may contain repeated elements. The formal definition is as follows:

- . Type definition
 

Type Bag(Element)                      [parameterized]  
 emptybag :  $\rightarrow$  Bag  
 put : Bag  $\times$  Element  $\rightarrow$  Bag  
 remove : Bag  $\times$  Element  $\rightarrow$  Bag  
 contains : Bag  $\times$  Element  $\rightarrow$  Bool
- . Set of axioms
 

For all b:Bag and for all e,e':Element,  
 remove(emptybag, e) = emptybag  
 remove(put(b, e), e') =  $e=e' \rightarrow b$  ; put(remove(b, e'), e)  
 contains(emptybag, e) = false  
 contains(put(b, e), e') =  $e=e' \vee$  contains (b, e')

These axioms are largely self-explanatory in the sense that they reflect a number of properties one expects to hold for "bags". The fact that the operations are completely defined by these axioms for all finite bags requires a little thought, but will also become clear from the following discussion.

#### d. A syntactic view (the "term model")

Let us rewrite the first three type definitions of the preceding example as a grammar:

```
bag = emptybag
    | put (bag, element)
    | remove (bag, element)
```

We define proper sentences as sentences not containing the symbol "remove". The axioms for "remove" can be used as transformation rules to convert any sentence into a proper sentence. It is easy to see that the intuitive interpretation is thereby not changed. "Contains" is now just a predicate on proper sentences. Hence the language defined in this way can be used as a semantic model for the data type definition. This model is called the "term model".

It is interesting to mention a recent paper [HEER86] in which this process is reversed; the concrete syntax of a programming language is converted into an abstract syntax that has the form of a type definition. For instance, let the concrete syntax be

```
if = if expr then series else series endif
while = while expr do series endwhile
stat = if | while
series = stat ; ... ; stat
```

then the abstract syntax is

```
if: Expr × Series × Series → Stat
while: Expr × Series → Stat
series: Stat* → Series
```

#### e. Discussion

- Questions regarding the choice of the set of axioms:
  - . Is it consistent (e.g. is  $\text{Bag} \neq \emptyset$ )?
  - . Is it complete (does it specify what we want)?
  - . Is it not "overspecific"?
- Issues regarding the meaning of the axioms:
  - . Generation from a basis element (e.g. `emptybag`) using a constructor function (e.g. `put`).
  - . Role of equality: do we introduce an explicit operation (e.g. `equal : Bag × Bag → Bool`) together with appropriate axioms, or do we establish a general convention in the definition language?

- . Existence of an induction principle over the constructor function, e.g.

$$P(\text{emptybag}) \wedge \underline{A}b:B, e:E. \{ [P(b) \Rightarrow P(\text{put}(b, e))] \} \Rightarrow \underline{A}b:B. P(b)$$

Do we add this as an additional axiom or refer to an overall convention?

- The degree of "abstraction".
  - . Abstraction is often illusory: the term model with the axioms seen as rewrite rules constitutes a very concrete view (a "prototype representation": easily derived though inefficient).
  - . Hence we prefer the constructive approach (against the purists) giving explicit expressions for the operations. An example is given in section 2.

#### f. Abstract an concrete data types

Concrete data types are introduced to realize abstract data types in the following way.

Consider an "abstract" operation  $f : A \rightarrow B$ .

Corresponding to the domain  $A$  and the range  $B$  (in this context called "denotation domains") we define "representation domains"  $A'$  and  $B'$  together with two mappings:

- . a representation mapping  $r : A \rightarrow A'$
- . a denotation mapping  $d : B' \rightarrow B$

The concrete operation  $c_f : A' \rightarrow B'$  is said to realize  $f$  if  $d \circ c_f \circ r = f$

Various applications of this simple paradigm in the area of digital systems are described in [BOUT86b]. The same paradigm also leads to some interesting reformulation of the theory of transforms in electrical engineering and signal processing.

## 2. FUNCTIONAL FORMALISMS

### 2.0 Declarative versus imperative formalisms

a. Illustration by means of an example from programming.

Assume one wishes to compute the sum of all numbers in a list.

- Imperative solution (the term "imperative" reflects that a program is a sequence of commands - in some language manuals incorrectly called "statements"):

```
function sum(b: array natural of number) returns number
  sum := 0; index := 0; n := length(b);
```

```

    while index<n do sum := sum + b(index); index := index + 1 od;
end function

```

This program illustrates the following characteristics of imperative languages:

- . Lack of expressive power, which is reflected by the need for auxiliary variables (index, n) and the changing meaning of the variables (e.g. "sum" designates the desired sum only at the exit from the loop). The last point necessitates the introduction of the concept of "state", which is the main reason for the next characteristic:
- . Lack of manipulative power, reflected by the fact that a separate proof system is necessary (pre- and postconditions) and program transformation is cumbersome.

Declarative solutions (so called because a "program" is a collection of statements - in the original and correct sense of the word).

Here we distinguish between two major styles:

- . The logic style: using predicates  
For instance, in Prolog [CLOC81]:

```

sum(0, [])
sum(S, B) :- first(X, B), rest(Z, B), sum(Y, Z), S is X+Y

```

Here the following observations hold:

Pro: the predicates are symmetric in the sense that "input" and "output" are interchangeable.

Contra: . Lack of expressive power, reflected by the need for many auxiliary "logic variables".

- . Lack of manipulative power, reflected by the need for a separate proof system and the subtle way in which the effect of the program depends on the execution sequence (back-tracking).

- . The functional style: using functions  
For instance, in SASL [TURN79]:

```

sum () = 0
sum x = (first x) + sum(rest x)

```

Here the following observations hold:

Contra: asymmetry (argument and result are not interchangeable)

Pro: . high expressive power (no excess nomenclature)

- . high manipulative power (proof by expression manipulation)

b. Advantages of functional formalisms in areas outside programming.

Functional formalisms provide a more "natural" style for system description, firstly because they are data-flow rather than control-flow oriented and hence better suited to reflect the transfer of information in physical or man-made systems; secondly because it is possible to achieve a closer match between language concepts and system concepts [BOUT84,86a].

Furthermore, functional formalisms yield valuable contributions to the formulation of the theory in several application domains, including not only digital systems as discussed in the two companion papers but also analog systems such as control and communication systems. One example is given in [BOUT85a].

Finally, functional formalisms are much closer to applied mathematics and the traditional "physical" engineering disciplines than is the case for logic formalisms. Indeed, the classical operator calculus over real and complex domains has a direct generalization in functional formalisms through the usage of higher-order functions over arbitrary domains (in particular, discrete domains). This fact provides a ready linkage between the disciplines.

## 2.1 Higher-order functions

### a. Definitions and conventions

A higher-order function is a function whose argument and/or result is itself a function. As usual, a function is defined by specifying the result of its application to an arbitrary argument (represented by a variable).

Example: consider the following definitions:

$$\text{sum} : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \quad \text{with} \quad \text{sum}(x, y) = x+y \quad (0)$$

$$\text{plus} : \mathbb{R} \rightarrow (\mathbb{R} \rightarrow \mathbb{R}) \quad \text{with} \quad (\text{plus } x) y = x+y \quad (1)$$

Definition (0) defines a "normal" (first-order) function whose argument is a pair of real numbers and whose result is a real number. Definition (1) defines a first-order function whose argument is a real number and whose result is a function of type  $\mathbb{R} \rightarrow \mathbb{R}$ . For instance, plus 3 is a function that adds 3 to its argument.

(end of example)

Following the convention established in subsection 0.3, we write  $fxy$  for  $(fx)y$  and  $X \rightarrow Y \rightarrow Z$  for  $X \rightarrow (Y \rightarrow Z)$ . Notice that we have already used a higher-order function in subsection 1.4 for the semantic definition of PL.

More generally, if the functions

$$f : (A_0 \times \dots \times A_{n-1}) \rightarrow A$$

$$g : A_0 \rightarrow \dots \rightarrow A_{n-1} \rightarrow A$$

satisfy  $f(a_0, \dots, a_{n-1}) = g \ a_0 \ \dots \ a_{n-1}$ , then we say that  $g$  is the Curried version of  $f$ . Notice that Currying is the most convenient way for defining functions that return functions as a result. The absence of this feature in LISP is the main reason why the potential of functional programming is only partially realized in that language.

#### b. Example: constructive definition of Bags

Recall the type definition:

```
Type Bag(Element)
  emptybag : → Bag
  put : Bag × Element → Bag
  remove : Bag × Element → Bag
  contains : Bag × Element → Bool
```

We now define Bag in a constructive way, replacing the axioms by the following definitions

```
Bag = Element → ℕ
emptybag e = 0
put (b, e) e' = (e' = e) → (b e' + 1) ; b e'
remove (b, e) e' = (b e = 0) ∨ (e' ≠ e) → b e' ; b e - 1
contains (b, e) = (b e > 0)
```

Purists would consider this definition as insufficiently abstract because the definition  $\text{Bag} = \text{Element} \rightarrow \mathbb{N}$  makes the type explicit and hence constitutes a "choice of representation". However, this type definition reflects the intuitive view of a bag (as described by an "inventory" of the elements it contains) more accurately than an axiomatic definition, and hence constitutes a more suitable level of abstraction.

Another interesting example is given in the Appendix.

## 2.2 Transformational reasoning

### a. Principle

The difference between the transformational style and the usual deductive style is outlined in the following table:

| issue of interest:     | reasoning style        |                            |
|------------------------|------------------------|----------------------------|
|                        | deductive              | transformational           |
| basic objects          | assertions             | expressions                |
| rules for reasoning    | logic axioms           | axioms of equality         |
|                        | inference rules        | substitution               |
|                        | theory-specific axioms | theory-specific equalities |
| flow of reasoning      | tree-structured        | linear                     |
| "flavour" of reasoning | logic                  | algebraic                  |

#### b. Example

In order to illustrate the transformational proof style, we present the correctness proof of the functional definition of a general Currying function, i.e. a function mapping an arbitrary unCurried function onto its Curried counterpart.

The Currying function is specified as follows:

$$\begin{aligned} \text{curry} : N::n \rightarrow (A_{n-1} \times \dots \times A_0 \rightarrow A) \rightarrow (A_{n-1} \rightarrow \dots \rightarrow A_0 \rightarrow A) \\ (\text{curry } n \ f) \ a_{n-1} \ \dots \ a_0 = f(a_{n-1}, \dots, a_0) \end{aligned} \quad (S)$$

Notice that the extra parameter  $n$  represents the only information that is needed about the type of  $f$  in order to achieve Currying. In current functional languages, it is not possible to extract this information from the argument  $f$ , hence it is supplied separately.

Let us now define a "functional program" [TURN79]:

$$\text{def } \text{curry } 0 \ f = f() \quad (0)$$

$$\text{curry } n \ f \ x = \text{curry } (n-1) \ (\text{pa } f \ x) \quad (1)$$

$$\text{where } \text{pa } f \ a \ x = f(a:x) \quad (2)$$

(clarification: in expressions involving tuples,  $:$  denotes prefixing. There is no danger of confusion with the different meaning of  $:$  occurring in type definitions)

#### Theorem

The definition (0..2) satisfies the specification (S)

Proof (by induction)

- . base step: for  $n=0$ , (S) and (0) are identical
- . induction hypothesis: take (S) as induction hypothesis for some  $n$
- . induction step (from  $n$  to  $n+1$ ):

$$\begin{aligned} (\text{curry } (n+1) \ f) \ a_n \ \dots \ a_0 \\ = \{\text{convention parentheses}\} (\text{curry } (n+1) \ f \ a_n) \ a_{n-1} \ \dots \ a_0 \end{aligned}$$

```

= {definition (1)} (curry n (pa f an)) an-1 ... a0
= {induction hypothesis} (pa f an)(an-1, ..., a0)
= {convention parentheses} pa f an (an-1, ..., a0)
= {definition (2)} f(an:(an-1, ..., a0))
= {definition :} f(an, an-1, ..., a0) (end of theorem)

```

Obviously the trivial steps corresponding to the convention with the parentheses would be omitted in a regular proof.

### 2.3 A tiny bit of theory

The theoretical basis of functional programming is the  $\lambda$ -calculus. We present a brief summary as cultural background and as a preparation for the companion paper on system semantics [BOUT86a].

#### a. Syntax of the $\lambda$ -calculus

```

term = variable
      | ( $\lambda$ variable.term)    ["abstraction"]
      | (term term)         ["application"]

```

#### b. Intuitive semantics

Abstraction turns an expression into a function by indicating which variable in the expression has to take (the value of) the argument. For instance, given the expression  $x+y$  then  $\lambda x.(x+y)$  is a function with one argument such that  $(\lambda x.(x+y))a = a+y$

#### c. Formal (syntactic) conversion rule: $\beta$ -conversion

$(\lambda v.M)N \leftrightarrow (M \text{ with } N \text{ properly substituted for every free occurrence of } v)$

The notation  $\leftrightarrow$  means that the left-hand side may be replaced by the right-hand side and vice versa, also "in context" (i.e. when it is a part of a larger term). The qualifications "properly" and "free" reflect precautions that are necessary to avoid confusion between variable names. The technical aspects are discussed in [STOY77].

Examples:  $(\lambda x.\lambda y.yx)z \leftrightarrow \lambda y.yz$

$(\lambda x.\lambda y.(\lambda x.xz)yx)z \leftrightarrow \lambda y.(\lambda x.xz)yz$

$(\lambda x.\lambda y.yx)y \leftrightarrow \lambda u.uy$

The first example is straightforward, the other two examples reflect the two kinds of precautions taken. "Blind" substitution would yield  $\lambda y.(\lambda x.zz)yz$  and  $\lambda y.yy$  respectively, which clearly has not the same meaning according to the intuitive semantics, nor according to the formal semantics outlined next.

#### d. Formal semantics

Since a "term" can be a "function" as well as an "argument" (e.g.  $x$  is a "function" and  $z$  an "argument" in  $xz$  but the roles are reversed in  $zx$ ), any single domain  $D$  that is used as the domain of interpretation must be reflexive, that is: must in some sense be equivalent to its function domain  $D \rightarrow D$ . Since for any nontrivial set (i.e. not a singleton)  $|D \rightarrow D| > |D|$ , this requires the restriction of  $D \rightarrow D$  to a suitable subset, viz. the set of continuous functions from  $D$  to  $D$ , written  $[D \rightarrow D]$ . Here "continuity" is meant in the lattice-theoretic sense [STOY77].

We then say that a domain  $D$  is reflexive if there exist continuous functions  $F : D \rightarrow [D \rightarrow D]$  and  $G : [D \rightarrow D] \rightarrow D$  such that  $F \circ G$  is the identity function on  $[D \rightarrow D]$ .

There are several ways to construct such a reflexive domain, the most important ones being the inverse limit construction [SAND73] and the  $P_\omega$  model [STOY77] (based on the powerset of the set of natural numbers).

Assuming that we have such a reflexive domain  $D$  as certified by the existence of the functions  $F$  and  $G$ , we now can define the semantics as follows:

$m : T \rightarrow I \rightarrow D$  where  $I = V \rightarrow D$   
 $m \ v \ i = i \ v$  for  $v : I$   
 $m \ [\lambda v.M] \ i = G(m' \ [\lambda v.M] \ i)$  using the auxiliary meaning function (defined for  
 abstractions only)  $m' : A \rightarrow I \rightarrow [D \rightarrow D]$  with  
 $m' \ [\lambda v.M] \ i \ d = m \ M \ i'$  where  $i'v' = (v' = v) \rightarrow d ; i \ v'$   
 $m \ [MN] \ i = F \ (m \ M \ i) \ (m \ N \ i)$

Theorem (consistency)  $\beta$ -conversion preserves meaning (end of theorem)

A more extensive treatment can be found in [STOY77]. Perhaps the most complete reference on the lambda calculus is [BARE81]. An alternative calculus using so-called combinators instead of variables and abstraction is described in [CURR74].

#### e. Correspondence with functional languages

The lambda calculus is a "purified" description of the way in which variables are substituted when elaborating function application. Two small steps suffice to make the correspondence with functional languages directly visible:

- The addition of constants representing predefined values and functions (e.g. numbers and arithmetic operations)
- The addition of some syntactic sugar for attaching a name to a function, viz. writing  $fx = \text{expr}$  for  $f \equiv \lambda x. \text{expr}$

A fairly complete treatment of the structure and implementation of functional languages and the role of lambda calculus and combinators in this respect is given in a forthcoming book by S. Peyton Jones.

#### 2.4 Application to system description

Whereas the application of functional languages as a replacement for imperative programming languages is well-known [PEYT86], their advantages as a description formalism in areas outside computer science are only beginning to emerge. In the two companion papers [BOUT86a,b], the application to the description of digital and analog circuits, mixed systems and computer architecture will be illustrated. Here we present only a small example.

Example: description of digital circuits

The following circuit definitions are written in FUNDS (Functional Description of Systems) [BOUT86a], whose basic syntax is inspired by the simplicity of SASL [TURN79].

Combinational circuit: a data selector

```
def select(s, d0, d1) = or(and(not s, d0), and(s, d1))
```

Sequential circuit: a divider by four

```
def divide in = and(q1, q0')
      feedback (q0, q0') = dff select(in, q0, q1')
      (q1, q1') = dff select(in, q1, q0)
```

According to the principle of system semantics, FUNDS descriptions have various interpretations, the most important one being:

- the structural interpretation as a circuit, where "and", "or", "not" stand for gates and "dff" for a d-type flipflop
  - the behavioural interpretation as a function describing the relationship between input and output sequences
- (end of example)

## 3. CONCLUSION

In this paper, we hope to have presented a concise yet understandable overview of the concepts we consider important as a basis for the formal description of embedded systems. The references are carefully selected (although unavoidably influenced by personal preference) to indicate the most appropriate sources of additional material on each particular topic.

## REFERENCES

- [BARE81] H.P. Barendregt, The lambda calculus: its syntax and semantics. North Holland, Amsterdam (1981)
- [BERG85] J.A. Bergstra, J.W. Klop, "Algebra of communicating processes", in: J.W. de Bakker et al., eds., Proc. CWI Symposium on Mathematics and Computer Science. CWI, Amsterdam (1985)
- [BOUT80] R.T. Boute, "Towards a theory of system semantics - an introductory overview of the key concepts for telecommunication engineers". Report RTB-8011, Bell Telephone Mfg. Cy., Antwerpen (Nov. 1980).
- [BOUT84] R.T. Boute, "System semantics applied to digital and analog circuits". Report No 56, Dept. of Computer Science, University of Nijmegen (Aug. 1984).
- [BOUT85a] R.T. Boute, "On the equivalence of time-division and frequency-division multiplexing". IEEE Comm 33, 1, pp. 97-99 (Jan. 1985).
- [BOUT85b] R.T. Boute, "Functional temporal calculus versus temporal logic: an engineering approach". Report No. 59, Dept. of Computer Science, University of Nijmegen (March 1985).
- [BOUT86a] R.T. Boute, "An introduction to system semantics", in: A. Kündig, ed., New Approaches to the Formal Description and the Design of Embedded Systems.
- [BOUT86b] R.T. Boute, "Current research topics in the functional approach to system description", in: A. Kündig, ed., New Approaches to the Formal Description and the Design of Embedded Systems.
- [CLEA77] J.C. Cleaveland, R.C. Uzgalis, Grammars for programming languages. Elsevier Programming Languages Series No. 4. Amsterdam (1977).
- [CLOC81] W.F. Clocksin, C.S. Mellish, Programming in Prolog. Springer, Berlin (1981).
- [CURR74] H.B. Curry, R. Feys, Combinatory logic. North-Holland, Amsterdam (1974).
- [EEKE86] M.C.J.D. van Eekelen, M.J. Plasmeijer, "Specification of reduction strategies in a TRS". Internal report, Dept. of Computer Science, University of Nijmegen (Jan. 1986).
- [GOGU78] J.A. Goguen et al., "An initial algebra approach to the specification, correctness and implementation of abstract data types", in: R.T. Yeh, ed., Current trends in programming methodology, vol IV: data structuring, pp. 80-149. Prentice-Hall, Englewood Cliffs, N.J. (1978)
- [HARR78] M.A. Harrison, Introduction to formal language theory. Addison-Wesley, Reading, Mass. (1978).
- [HEER86] J. Heering, P. Klint, "User definable syntax for specification languages". Proc. NGI-SION 1986 Symposium, pp. 63-73 (April 1986).
- [KALD86] A. Kaldewaij, A formalism for concurrent processes. Ph. D. Thesis, Eindhoven University of Technology (May 1986).
- [KLOP85] J.W. Klop, "Term Rewriting Systems", Seminar on reduction machines, Ustica (Sept. 1985).
- [KOST71] C.H.A. Koster, "Affix grammars", in: J.E.L. Peck, ed., Algol 68 implementation, pp. 95-109. North-Holland Publ. Co., Amsterdam (1971).

- [LEVI74] M. Levin, "Mathematical logic for computer scientists". Report MAC TR-131. Massachusetts Institute of Technology (June 1974).
- [MEIJ86] H. Meijer, Programmar: a translator generator. Ph. D. Thesis, Dept. of Computer Science, University of Nijmegen (June 1986).
- [MEND79] E. Mendelson, Introduction to mathematical logic, 2nd ed. Van Nostrand, New York (1979).
- [MILN80] R. Milner, A calculus of communicating systems (LNCS 92). Springer, Berlin (1980).
- [PEYT86] S.L. Peyton Jones, "Functional programming languages as a software engineering tool", in: A Kündig, ed., New Approaches to the Formal Description and the Design of Embedded Systems.
- [REMM85] M. Rem, "Concurrent computations and VLSI circuits", in: M. Broy, ed., Control flow and data flow: concepts of distributed programming, pp. 399-437. Springer, Berlin (1985).
- [SAND73] J.G. Sanderson, The lambda calculus, lattice theory and reflexive domains, Mathematical Institute Lecture Notes, University of Oxford (1973).
- [STAN78] T.A. Standish, "Data structures - an axiomatic approach", in: R.T. Yeh, ed., Current trends in programming methodology, vol IV: data structuring, pp. 30-59. Prentice-Hall, Englewood Cliffs, N.J. (1978).
- [STOY77] J.E. Stoy, Denotational semantics: the Scott-Strachey approach to programming language theory. The MIT Press, Cambridge, Mass. (1977).
- [TURN79] D.A. Turner, SASL language manual (revised version). University of Kent (1979).
- [TURN85] D.A. Turner, "Miranda: a non-strict functional language with polymorphic types", in: J.P. Jouannand, ed., Functional programming languages and computer architecture, pp. 1-16. LNCS 201, Springer, Berlin (1985).
- [WIRTH77] N. Wirth, "What can we do about the unnecessary diversity of notation for syntactic definitions?", Comm. ACM 20, 11, pp. 822-823 (Nov. 1977).
- [WULF76] W.A. Wulf, R.L. London, M. Shaw, "An introduction to the construction and verification of Alphard programs", IEEE SE-2, 4, pp. 253-265 (Dec. 1976).

## APPENDIX: Data type abstraction in MIRANDA

The following example is taken from [TURN85]. It illustrates how the system of propositional logic (see section 1.4) can be described as an abstract data type.

```
wff ::= var[char] | wff $implies wff | not wff
```

```
abstype theorem
```

```
with axiom1, axiom3 :: wff→wff→theorem;
```

```
    axiom2 :: wff→wff→wff→theorem;
```

```
    modus_ponens :: theorem→theorem→theorem;
```

```
    contents :: theorem→wff;
```

```
theorem == wff
```

```
axiom1 a b = a $implies (b $implies a)
```

```
axiom2 a b c = (a $implies (b $implies c)) $implies ((a $implies b) $implies
    (a $implies c))
```

```
axiom3 a b = not(a $implies b) $implies (not b $implies not a)
```

```
modus_ponens a (a $implies b) = b
```

```
contents x = x
```

# AN INTRODUCTION TO SYSTEM SEMANTICS

Raymond T. Boute

Department of Computer Science

University of Nijmegen, The Netherlands

## ABSTRACT

System semantics is the description of system properties by means of semantic functions. For a given formal description of the considered system, different semantic functions describe different properties of interest. As a formalism for language definition, system semantics also provides very useful guidance in the design of system description languages (SDLs), in particular the matching of language constructs to system concepts in a way that yields useful formal properties for reasoning and formal manipulation.

This paper explains the basic formalism of system semantics and illustrates the application to the description of digital circuits and analog circuits. The existence of useful formal properties is illustrated by showing that theorems of switching theory or electrical network theory can be expressed as conversion rules in the language.

## 0. INTRODUCTION

System semantics has a twofold motivation:

- the unsatisfactory state of the art in the design of system description languages (SDLs),
- the promise offered by functional formalisms in system description and analysis.

We briefly elaborate on the first point, the second point being illustrated throughout the remainder of this paper.

As is apparent from the very good treatment in [STAB70] (which, although not recent, is still representative for today's state of the art), most system description languages clearly exhibit their imperative heritage from early programming languages in the usage of assignment to describe interconnections and of repetition to describe multiple instantiations. In fact, most SDLs are simulation languages (to be used as input for a simulator) rather than description languages in the proper sense.

In the area of digital systems, there has been some improvement in recent years, as can be seen from

- the more judicious usage of imperative constructs (providing them with more descriptive semantics) as in Zeus [GERM85],
- the emergence of descriptive SDLs of the logic family [BARR84, MARU85, SUZU85],
- the emergence of descriptive SDLs of the functional family [JOHN84, SHEE84],

but a general guiding principle is still missing.

In the area of analog systems, the situation is considerably worse. Some SDLs are static but highly unstructured, e.g. describing interconnections by node identifiers [DIGI82]. Other notations are more structured, e.g. in terms of two-ports [GUPT81, TYMS72], but then again use imperative constructs. In this situation, defining formal semantics or establishing useful properties for formal manipulation is impossible for all practical purposes.

It will be seen that system semantics opens the way for substantial improvements. Whereas this paper is only an introductory overview, a more complete treatment can be found in [BOUT85]

## 1. PRINCIPLES OF SYSTEM SEMANTICS

### 1.0 Formal language specification

Assuming that the reader is familiar with the material in [BOUT86], we can be very brief on this point.

#### a. Syntax

Syntax is defined by means of a grammar using (simplified) BNF for the context-free part and explicit restrictions for the context-sensitive part. We reserve the following letters for syntactic categories: S (sentences), C (constants), V (variables).

#### b. Semantics

Semantics is defined by means of a model consisting of a domain of interpretation  $D_X$  for every important syntactic category X and meaning functions of the form

$m_X : X \rightarrow I \rightarrow D_X$  where  $I = V \rightarrow D_V$

or, in case the meaning is independent of the interpretation for the variables,

$m_X : X \rightarrow D_X$  (e.g. for sentences).

### c. Equivalence and conversion rules

Semantic equivalence with respect to a model  $M$  is defined by a relation  $=_M$  on sentences as follows:

$$s =_M r \Leftrightarrow m s = m r$$

Conversion rules are syntactic relations (written  $\leftrightarrow$ ) between sentences.

Syntactic equivalence with respect to a set of conversion rules  $R$  is defined by a relation  $=_R$  on sentences as follows:

$$s =_R r \Leftrightarrow s = r \quad (\text{reflexive closure}).$$

$$\text{or } r =_R s \quad (\text{symmetric closure})$$

or there is a sentence  $t$  such that

$$r =_R t \text{ and } t =_R s \quad (\text{transitive closure})$$

or  $r \leftrightarrow s$  for some rule  $\leftrightarrow$  in  $R$

Stated informally: two sentences are syntactically equivalent if one can be transformed into the other in a finite number of steps, using one of the conversion rules at every step. For the languages of interest here, we also postulate that  $=_R$  is a (2-sided) congruence relation by adding the extra condition

$$s =_R r \Leftrightarrow asb =_R arb$$

i.e. conversion rules may also be used to transform parts of sentences independently of the context.

With respect to a model  $M$ , a set of conversion rules  $R$  is called

- consistent (or meaning-preserving) if  $r =_R s \Rightarrow r =_M s$
- complete (or sufficiently powerful) if  $r =_M s \Rightarrow r =_R s$

## 1.1 System semantics

### a. Principle

Given a single SDL for the considered class of systems, we define a different semantic model for every property of interest. The domain of interpretation is the set of "values" for the considered property.

### b. Relation between models

We say that model  $M$  is at least as detailed as model  $M'$ , written  $M \geq M'$ , if  $r =_M s \Rightarrow r =_{M'} s$ . A strictly initial model is a model that is isomorphic to the grammar, i.e.  $M$  is initial if  $r =_M s \Rightarrow r = s$ . Notice that  $(M \text{ initial}) \Leftrightarrow (M \geq M' \text{ for all } M')$ . Considering only the models of practical interest, we define an initial model as a model that is at least as detailed as any model of practical interest.

### c. Housekeeping measures

When dealing with many different models, it is useful to reduce the writeup of the definitions by factorizing out common parts. This is done at two levels:

- At the semantic level: for a class of models that are "related" in the sense that they have a common definition part, this common part is written separately as a generic part. The models in that class are then said to be conform to the generic part, and only the model-specific parts have to be defined (usually involving only the interpretation of the constants). Models that constitute a class of their own are called singular.
- At the syntactic level: the introduction of "syntactic sugar" and of new constructs by means of macro definitions.

## 1.2 Language design issues

A good language design criterion for ensuring useful formal properties is compositionality, i.e. the meaning of a composition of components depends only on the meaning of the composition rule and on the meanings (excluding all other information) of the components. Written somewhat more formally:

$$m \llbracket \text{comp}(\text{part}_0, \dots, \text{part}_{n-1}) \rrbracket = f(m \text{ comp}, m \text{ part}_0, \dots, m \text{ part}_{n-1}).$$

Obtaining compositionality requires a careful match between language concepts (or constructs) and system concepts at two levels:

- a. Choice of the semantic basis: imperative, declarative (functional, logic), etc.
- b. Detailed association of syntactic categories with system concepts.

In this paper, (a) is resolved by selecting a functional basis, and (b) by the following association:

| "order" or category | language concept | system concept |
|---------------------|------------------|----------------|
| 0                   | variables        | interfaces     |
| 1                   | functions        | subsystems     |
| 2                   | combinators      | (patterns)     |

It turns out that the main issue determining compositionality is the flow of information inside the system, characterized by:

- unidirectional vs. bidirectional components,
- loading characteristics of the components (output drive, input load).

## 2. APPLICATION TO UNIDIRECTIONAL SYSTEMS: DIGITAL CIRCUITS

The language concepts introduced in this paper constitute the basis for a functional system description language called FUNDS (FUNctional Description of Systems).

### 2.0 Syntax (combinational subset of FUNDS)

The free part of the abstract syntax is described as follows:

```

function = constant | abstraction
expression = variable | application
constant = zero | one | and | or | not
variable = a | a0 | a1 | ... | z | z0 | z1 | ... | a' | a0' | ...
abstraction =  $\lambda$ (variable1 ... variablen).expression
application = function(expression1 ... expressionn)

```

Restrictions are twofold: the variables in the  $\lambda$ -part of an abstraction must be distinct, and the number of expressions in an application must match the number of places associated with the function. For the constants, the number of places is given, whereas for an abstraction the number of places is the number of variables in the  $\lambda$ -part. Variables are by definition zeroth-order objects. The main syntactic categories introduced by this grammar are F (functions), E (expressions), C (constants), V (variables) and S (sentences). The sentences are defined to be the abstractions containing bound variables only.

The concrete syntax is inspired by SASL [TURN79]. To introduce a name  $f$  for a sentence of the form  $\lambda(v_0, \dots, v_{n-1}).e$  we shall write

def  $f(v_0, \dots, v_{n-1}) = e$

Furthermore, an application of the form  $\lambda(v_0, \dots, v_{n-1}).e(e_0, \dots, e_{n-1})$  will be written

$e$  where  $(v_0 = e_0, \dots, v_{n-1} = e_{n-1})$

Example: def  $\text{select}(s, d0, d1) = \text{or}(\text{and}(\text{not } s, d0), \text{and}(s, d1))$

## 2.1 Overview of the models (for the combinational subset of FUNDS)

### a. Generic semantic model

#### - Model-dependent part.

Let  $D$  be the zeroth-order domain of interpretation, then the interpretation for the constants is of the general form  $k : C \rightarrow D_C$  where  $D_C = D^* \rightarrow D$ . More specifically, for an  $n$ -place constant  $c$ ,

$k \ c : D^n \rightarrow D$  (remark:  $D^* = \bigcup_{n:\mathbb{N}} D^n$ ).

#### - Generic part

The meaning function for expressions, functions and sentences are:

$\text{mexp} : E \rightarrow I \rightarrow D$  with

$\text{mexp } v \ i = i \ v$  for variables, whereas for applications

$\text{mexp } [\![f(e_0, \dots, e_{n-1})]\!] \ i = \text{mfun } f \ i \ (\text{mexp } e_0 \ i, \dots, \text{mexp } e_{n-1} \ i)$

$\text{mfun } f \ i = F \rightarrow I \rightarrow D^* \rightarrow D$  with

$\text{mfun } c \ i = k \ c$  for constants, whereas for abstractions

$\text{mfun } [\![\lambda(v_0, \dots, v_{n-1}).e]\!] \ i : D^n \rightarrow D$  with

$\text{mfun } [\![\lambda(v_0, \dots, v_{n-1}).e]\!] \ i \ (d_0, \dots, d_{n-1}) =$   
 $\text{mexp } e \ i[d_0/v_0] \dots [d_{n-1}/v_{n-1}]$

$m : S \rightarrow D^* \rightarrow D$  with  $m \ s = \text{mfun } s \ i$  (where  $i$  is any interpretation)

The postfix operator  $[d/v] : I \rightarrow I$  is defined by

$i[d/v] \ v' = (v' = v) \rightarrow d; i \ v'$

The reader should devote some thinking to the differences between this model and the model for the lambda-calculus as given in [BOUT86a], in particular why no reflexive domain has to be introduced here.

## b. Models of interest

As usual, the syntax itself is a model for the language. (0)

Models conform to the generic model are:

- The structural model (with object semantics) (1)

- The behavioural models

- . (simplex) behavioural model with  $D = \{0,1\}$  (2)

- . sequence model with  $D = \{0,1\}^*$  (3)

- . discrete time function model with  $D = \mathbb{N} \rightarrow \{0,1\}$  (4)

- The hazard model with  $D = \{0, 1, x\}$  (5)

- based on Eichelberger algebra [EICH65]

- The worst-case timing model with  $D = \mathbb{R}_{\geq 0}$  (6)

An interesting singular model is the component cost model (7)

## c. Relation between the semantic models

For the various models mentioned under (b) we have

$$(0) \equiv (1) > (5) > (2) \equiv (3) \equiv (4)$$

$$> (6)$$

$$> (7)$$

where the ordering relation refers to the level of detail. As usual,  $M \equiv M'$  means  $M \geq M'$  and  $M' \geq M$ , whereas  $M > M'$  means  $M \geq M'$  but not  $M \equiv M'$ .

Since this is an introductory treatment, only the structural and behavioural models will be outlined.

## d. Structural model

The domain of interpretation  $D$  is the set of loop-free single-output circuits. The interpretation of the constants  $k: C \rightarrow D^* \rightarrow D$  is defined as follows:

$k \ c \ (d_0, \dots, d_{n-1})$  is the composition of the circuits  $d_0, \dots, d_{n-1}$  and a c-gate whose inputs are connected to the outputs of  $d_0, \dots, d_{n-1}$ . The inputs of the entire circuit are the inputs of  $d_0, \dots, d_{n-1}$ , and the output of the entire circuit is the output of the c-gate.

The meaning of functions and expressions is given in the generic model. There are no nontrivial conversion rules, as is expected for an initial model.

Notice that the where-construct allows to express fanout, as can be seen from comparing the definitions:

def  $ex(a, b, c) = or(and(not \ a, b), and(not \ a, c))$

def  $ex'(a, b, c) = or(and(a', b), and(a', c))$  where  $a' = not \ a$

The first definition implies the usage of two separate inverters, the second definition implies only one inverter but with fanout.

#### e. Simplex behavioural model

The domain of interpretation is  $D = \{0, 1\}$ . The interpretation for the constants  $k$  :  $C \rightarrow D^* \rightarrow D$  is given by

$k \text{ zero} : \rightarrow D$       with       $k \text{ zero} = 0$   
 $k \text{ one} : \rightarrow D$       with       $k \text{ one} = 1$   
 $k \text{ not} : D \rightarrow D$       with       $k \text{ not } d = \bar{d}$   
 $k \text{ and} : D^* \rightarrow D$       with       $k \text{ and } (d_0, \dots, d_{n-1}) = d_0 \wedge \dots \wedge d_{n-1}$   
 $k \text{ or} : D^* \rightarrow D$       with       $k \text{ or } (d_0, \dots, d_{n-1}) = d_0 \vee \dots \vee d_{n-1}$

The meaning for functions and expressions is given in the generic model.

There are two kinds of conversion rules:

- Conversion rules corresponding to the generic part (inherited from the lambda-calculus):

$\alpha$ -conversion: renaming bound variables, e.g.

def impl (a, b) = or(not a, b)

def impl (x, y) = or(not x, y)

$\beta$ -conversion: substitution

$\eta$ -conversion (not of interest here)

- Conversion rules corresponding to the constants.

These rules are a syntactic reformulation of the axioms of switching algebra (without quantifiers!). Let  $e_0, \dots, e_{n-1}$  be expressions,  $c : \{\text{and, or}\}$  and  $p$  any permutation on  $0..n-1$ , then these rules are:

$c(e_0, \dots, e_{n-1}) \leftrightarrow c(e_{p0}, \dots, e_{p(n-1)})$  (Commutativity)

$c(e_0, \dots, e_{n-1}) \leftrightarrow c(e_0, \dots, e_{k-1}, c(e_k, \dots, e_{n-1}))$  (Associativity)

$\text{and}(e_0, \text{or}(e_1, e_2)) \leftrightarrow \text{or}(\text{and}(e_0, e_1), \text{and}(e_0, e_2))$  (Distributivity and)

$\text{or}(e_0, \text{and}(e_1, e_2)) \leftrightarrow \text{and}(\text{or}(e_0, e_1), \text{or}(e_0, e_2))$  (Distributivity or)

$\text{and}(e, \text{one}) \leftrightarrow e, \text{or}(e, \text{zero}) \leftrightarrow e$  (Unit element and, or)

$\text{and}(e, \text{not } e) \leftrightarrow \text{zero}, \text{or}(e, \text{not } e) \leftrightarrow \text{one}$  (Inverse element and, or)

## 2.2 Extensions

### a. Extension with tuples (for multiple-output circuits)

This extension is left as a (simple but nontrivial) exercise to the reader. It will be assumed to be in effect henceforth.

## b. Extension to sequential circuits

### - Syntax extension (sequential FUNDS)

constant = memorycell

memorycell = dff | tff | jkff | rsff [flipflop types]

application = expression feedback fblist

fblist = variable = expression, ..., variable = expression

The constants dff and tff are one-place, jkff and rsff are two-place. All these constants have 2 output places (pairs).

Example: divide-by-four circuit

```
def divfour in = and(q1, q0')
```

```
    feedback (q0, q0') = dff select(in, q0, q1')
```

```
    (q1, q1') = dff select(in, q1, q0)
```

### - Semantic extension:

. Extension of the generic model for "feedback".

. Extension of the interpretation for constants.

For the behavioural semantics, the simplex model is trivial; only the sequence model and the discrete time function model are of practical interest (not elaborated here).

## c. Macro definitions

The purpose of macro definitions is twofold:

- as an abstraction mechanism (improving style),
- to introduce new language constructs (without having to extend the semantic models).

The syntax of a macro definition is

```
mac mname (v0 , ..., vn-1) = mbody
```

Macro expansion is done by  $\beta$ -conversion, i.e.

```
mname (exp0, ..., expn-1)
```

is replaced by mbody, with exp<sub>0</sub>, ..., exp<sub>n-1</sub> properly substituted for v<sub>0</sub>, ..., v<sub>n-1</sub>.

## d. Extended example

All the concepts introduced thus far are brought together in the following description of a SN74163 synchronous 4-bit counter (which can be compared to the schematic diagram of fig.0):

```

def counter (d3, d2, d1, d0, nload, nclear, enap, enat)
  = (carry, q3, q2, q1, q0) where carry = and(q3, q2, q1, q0, enat)
    feedback q3 = stage(d3, and(q2, q1, q0, en))
    q2 = stage(d2, and(q1, q0, en))
    q1 = stage(d1, and(q0, en))
    q0 = stage(d0, en)
    mac stage(d, toggle) = jkff(and(ja, ea), and(ka, ea))
    where ja = nand(ka, preset)
    ka = nand(d, nclr, preset)
    ea = or(toggle, preset)

where preset = ornot(nload, nclr)
    nclr = buf nclear
    en = and(enap, enat)

```

Notice the judicious usage of bound variables (d, toggle) and free variables (nclr, preset) in the macro definition to represent indirectional and common inputs respectively.

### 3. APPLICATION TO BIDIRECTIONAL SYSTEMS: ANALOG CIRCUITS

#### 3.0 Basic approach

##### a. Two-port circuits

Although in general a circuit is an arbitrary interconnection of 2-, 3- or n-terminal devices, we view it as an interconnection of 4-terminal devices called two-ports. For a large class of circuits, the reduction in generality is small, and sufficiently compensated by the advantages of the imposed structuring in view of analysis and design [GUPT81, NORD83]. The labelling conventions are shown in fig.1.

##### b. Syntax, generic model and structural model

The syntax of TPL (two-port language) is defined by

```

two-port = device          [primitive components]
          | combinator(twoport1 ...n twoportn)
combinator = inv | rev | p0p1 | p0s1 | cas | s0p1 | s0s1

```

together with the usual restriction on the number of places (inv and rev are one-place, the others have an arbitrary number of places). Syntactic categories are T (twoports), P (primitive components) and C (combinators). The language is extended

with a where construct to be viewed as a macro definition.

All models are conform to the following generic model (assuming domain of interpretation D):

- . interpretation for primitive components:  $k : P \rightarrow D$
- . interpretation for combinators:  $k' : C \rightarrow D^n \rightarrow D$
- . meaning of twoports:  $m \ p = k \ p$  for  $p:P$   

$$m \llbracket c(t_0, \dots, t_{n-1}) \rrbracket = k'c \ (m \ t_0, \dots, m \ t_{n-1})$$

The rationale behind these definitions is best illustrated by the structural model, where D is the set of two-ports built from smaller two-ports using combinators. The interpretation for the constants is:

- $k \ p =$  [primitive device designated by p]
- $k' \ inv \ d =$  the two-port d with upper and lower terminals interchanged
- $k' \ rev \ d =$  the two-port d with 0-port and 1-port interchanged
- $k' \ cascade \ (d, d') =$  the two-ports d and d' in cascade
- $k' \ p0p1 \ (d, d') =$  the two-ports d and d' with the 0-ports and the 1-ports connected in parallel

and so on.

### 3.1 General behavioural model and simplifications

#### a. General behavioural model

Of the eight parameters shown in fig.1, only six are independent due to Kirchoff's laws. We choose the 3 voltages  $e_0, e_1, e$  and the 3 currents  $i_0, i_1, i$  ( $i = i_0$ ). These parameters are time-varying functions of type  $\mathbb{R} \rightarrow \mathbb{R}$ . Since the behaviour of a two-port can be fully characterized by a relation among these parameters, the most general domain of interpretation is

$$D = (\mathbb{R} \rightarrow \mathbb{R})^6 \rightarrow \{\text{true}, \text{false}\}$$

For what concerns the interpretations for the constants,

$k : P \rightarrow D$  expresses laws for primitive devices,

$k' : C \rightarrow D^n \rightarrow D$  expresses Kirchoff's laws for interconnections.

Details can be found in [BOUT85]. Since the result of evaluating the meaning function is precisely the set of equations for the entire circuit, the tractability depends on the state of the art in circuit analysis. In the following two steps, we sacrifice generality in favour of tractability.

### b. First simplification step: typification

It is common practice to characterize two-ports by only 4 parameters ( $e_0, e_1, i_0, i_1$ ). However, the role of these parameters depends on the internal structure of the two-port, reflected by the following classification:

- N-type (natural) two-ports: the two ports are galvanically separated (e.g. transformers, gyrators, microwave twoports)
- T-type (top) two-ports: terminals 01 and 11 are connected
- $\perp$ -type (bottom) two-ports: terminals 00 and 10 are connected
- I-type (insert) two-ports: intersection of T and  $\perp$  type
- M-type (improper) two-ports: 4 parameters do not suffice

Typically T and  $\perp$  type two-ports are encapsulations of 3-terminal devices, and I-type two-ports are encapsulations of 2-terminal devices. For two-ports that are not improper, the 4 parameters together with the type fully describe the behaviour.

The typing can be described by means of a type model with domain of interpretation  $D = \{N, T, \perp, I, M\}$ . The definition is left as an exercise or can be found in [BOUT85].

### c. Second simplification step: specialized models

In practice, circuit analysis is not done on the basis of a single all-encompassing set of equations, but on the basis of a subdivision into separate aspects that can be analyzed independently or at least as subsequent refinements (perturbations). Typical examples are:

- DC behaviour (quiescent points)
- Small-signal behaviour (linearized)
- Distortion behaviour (perturbation analysis)
- Large-signal behaviour (non-linear analysis)
- Noise behaviour (statistical analysis)

All these aspects can be described by a suitable semantic model. For instance, the linear circuit model is derived as follows from the general model:

- The 4 time functions  $e_0, e_1, i_0, i_1$ , are replaced by their Fourier transforms

$$E_0, E_1, I_0, I_1,$$

- Behaviour is expressed by the reverse chain matrix equation:

$$\begin{bmatrix} E_0 \\ I_0 \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E_1 \\ -I_1 \end{bmatrix}$$

This leads to the following definition for the domain of interpretation:

$D = \text{Matrix} \times \text{Type}$  where

Matrix = Expr<sup>4</sup> + {undefined}

Type = {N, T, L, I, M}

Expr is the language of formal expressions in  $j\omega$  (describing Fourier transforms).

The definitions of the interpretation functions for the constants is left as an exercise or can be found in [BOUT85].

### 3.2 Example: formal definition, derivation of properties

As an example, we choose an RC phase-shift oscillator. The AC equivalent diagram (from [WEST72]) is shown in fig.2. The formal description is as follows

def osc = feedback shifter

where shifter = casc(cr, cr, S C C, S R R, Tce  $\beta$ , P R R<sub>L</sub>)

where cr = casc(S C C, P R R)

feedback x = p0p1(x, S R O)

Since we have not previously defined the sublanguage P of primitive devices, we briefly clarify:

S C C stands for "Series Capacitor with value C"

P R R stands for "Parallel Resistor with value R"

Tce  $\beta$  stands for "Transistor (common emitter) with gain  $\beta$ "

Hence the formal description defines the oscillator as feedback applied to a phase-shift network, both defined as macro's in a where clause.

Derivation of the oscillation condition is based on the following general theorem:

#### Theorem

feedback Dev(a, b, c, d, t) =<sub>B</sub> P Y (a + d - (ad - bc) - 1)/b

Clarification:

- . Dev(a, b, c, d, t) is a sentence in P defining a primitive device with given chain parameters and given type
- . =<sub>B</sub> indicates behavioural equivalence
- . P Y Y stands for "Parallel admittance with value Y"

Collary: [under very general assumptions] the oscillation condition for

Dev(a, b, c, d) with feedback is  $a + d - (ad - bc) = 1$

The proof of the theorem is done by evaluating the meaning of both sides of =<sub>B</sub> (see [BOUT85]). The application of the corollary to the RC phase-shift oscillator yields the oscillation condition:

$$\omega_0 = C^{-1}(4RR_L + 6R^2)^{-1/2} \quad (\text{oscillation frequency})$$

$$\beta = 23 + 29R/R_L + 4R_L/R \quad (\text{transistor gain})$$

As is also customary in traditional circuit analysis, the nonlinear effects determining the oscillation amplitude are ignored in this model, but can be introduced by means of a refinement.

#### 4. CONCLUDING REMARKS

##### a. Schematics versus formal description

Representing circuits by formulas (as for the RC oscillator example) instead of schematics has the following advantages:

- Expressive power

Schematics still require extra symbols and labels for external referencing (voltage and current arrows with letters corresponding to letters in the text M in equations). Formal descriptions by means of formulas are self-contained.

- Manipulative power

To analyze a circuit, given a schematic, formulas have to be derived anyway. Formal descriptions by means of formulas can be manipulated directly as in algebra.

Of course, schematics have a long-standing tradition and hence human factors play an important role. However, the situation is not unlike the usage of pictures in classical geometry versus the usage of formulas in analytic geometry.

##### b. Future work includes the following activities:

- Design and realization of a "system description environment" based on a functional language and catering for user-supplied semantic definitions.
- Elaboration of system semantics for digital systems: expressing detail below the structural level (e.g. layout), timing analysis, asynchronous circuits, description of bidirectional bus systems, etc.
- Elaboration of system semantics for analog systems: other models (DC, large signal, distortion, noise), structural generalization (beyond two-ports).

#### REFERENCES

- [BARR84] H.G. Barrow, "VERIFY: a program for proving correctness of digital hardware designs", Artificial Intelligence 24, pp. 411-436 (sept 1984)
- [BOUT85] R.T. Boute, "System semantics and formal circuit description". Report no. 69, Dept. of Computer Science, University of Nijmegen (Aug. 1985)

- [BOUT86] R.T. Boute, "Elements for the formal description of systems". In: A. Kündig, ed., New Approaches to the Formal Description and the Design of Embedded Systems
- [DIGI82] Digital Equipment Corp., "Spice2 circuit simulation", Engineering systems (brochure) (Oct. 1982)
- [EICH65] E.B. Eichelberger, "Hazard detection in combinational and sequential circuits", IBM Journal Res. and Dev., pp. 90-99 (March 1965)
- [GERM85] S.M. German, K.J. Liebherr, "Zeus: a language for expressing algorithms in hardware". IEEE Computer 18, 2, pp. 55-65 (Feb. 1985).
- [GUPT81] K.C. Gupta, R. Garg, R. Chadha, Computer aided design of microwave circuits. Artech House Inc., Dedham, Mass (1981)
- [JOHN84] S.D. Johnson, Synthesis of digital designs from recursion equations. The MIT Press, Cambridge, Mass. (1984)
- [MARU85] F. Maruyama, M. Fujita, "Hardware verification". IEEE Computer 18, 2, pp. 22-32 (Feb. 1985)
- [NORD83] E.H. Nordholt, Design of high-performance negative feedback amplifiers. Elsevier, Amsterdam (1983)
- [STAB70] E.P. Stabler, "System description languages", IEEE Trans. Comp. C-19, 12, pp. 1160-1173 (Dec. 1970)
- [SUZU85] N. Suzuki, "Concurrent Prolog as an efficient VLSI design language". IEEE Computer 18, 2, pp. 33-40 (Feb. 1985)
- [TYMS72] Tymshare, Inc., COMPACT (COMputerized optimization of Passive and ACTive circuits)
- [WEST72] H.P. Westman, ed., Reference data for radio engineers (ITT publication), Howard W. Sams & Co. Ltd., Indianapolis (1972)

## FIGURES

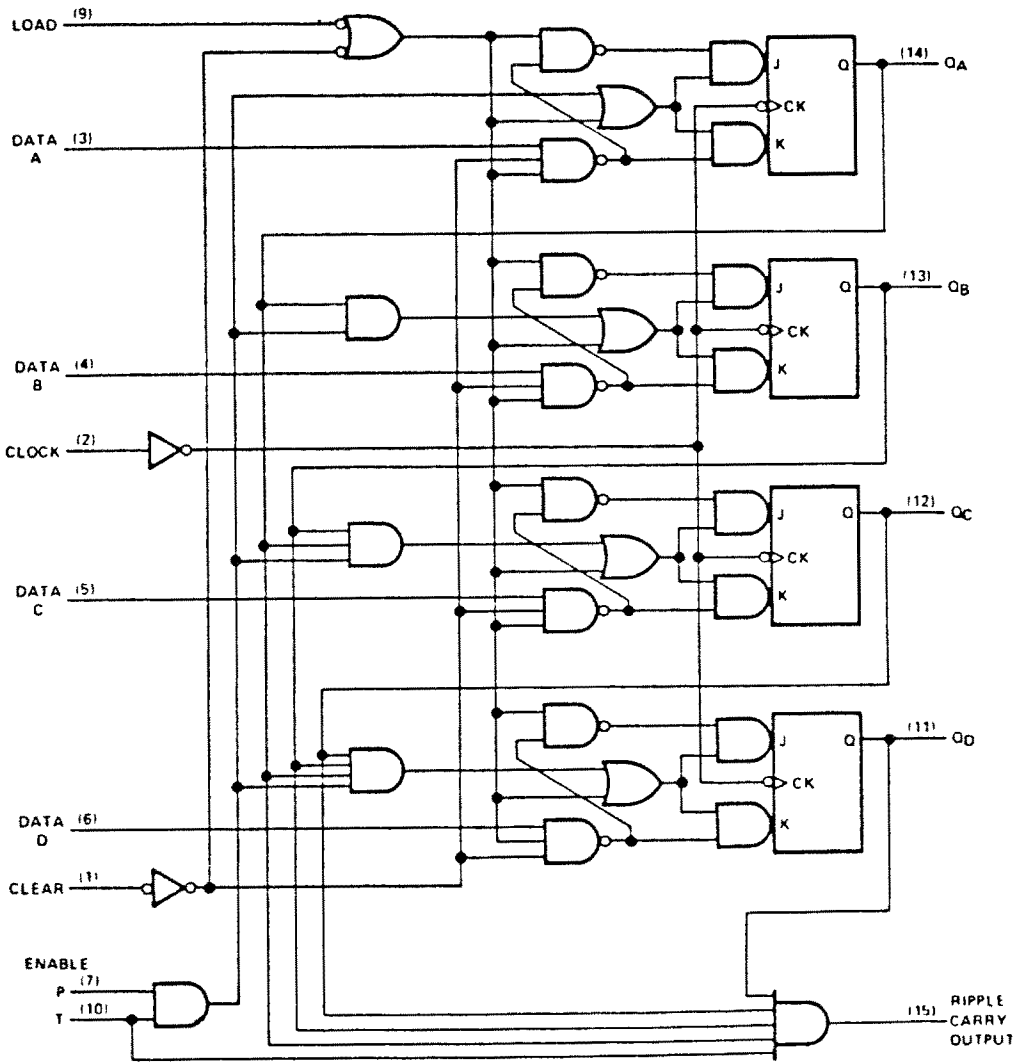


Fig. 0 SN74163 Synchronous Binary Counter

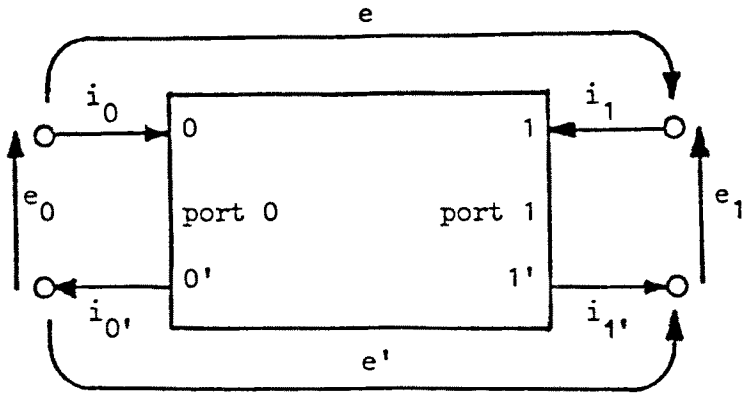


Fig. 1 Terminal labelling conventions

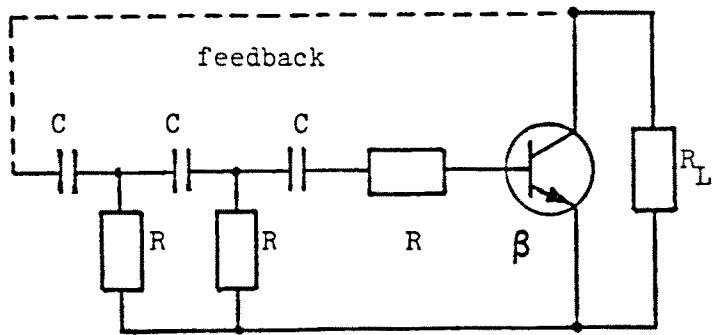


Fig. 2 RC phase-shift oscillator (AC equivalent)

FUNCTIONAL FORMALISMS FOR SYSTEM DESCRIPTION:  
CURRENT RESEARCH TOPICS

Raymond T. Boute  
Dept. of Computer Science  
University of Nijmegen, The Netherlands

ABSTRACT

This paper presents a cursory overview of current (and hence unfinished) research work on the application of functional formalisms to the description of systems. Following some introductory comments on their interrelation, the following topics are briefly discussed:

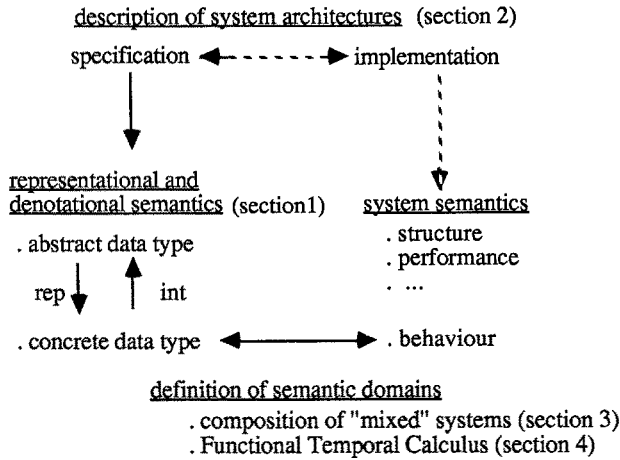
- Representational and denotational semantics as a paradigm for viewing physical systems as realizations of abstract operations,
- Functional description of computer architectures,
- Functional description of mixed systems (embedded analog and digital systems),
- The Functional Temporal Calculus as a general formalism for describing time-varying phenomena.

0. INTRODUCTION - RELATION BETWEEN THE TOPICS

The eventual objective of any formalism for system description is to cover all relevant degrees of abstraction, from specification to implementation, and all levels of the system hierarchy, from the total system with all its complexity to the individual primitive component. Furthermore, at every level of the system hierarchy, the correspondence between descriptions of various degrees of abstraction should be readily expressible. The topics discussed in this paper can be fitted into this context as shown in the diagram on the next page.

From top to bottom, this diagram shows decreasing levels of system hierarchy, and from left to right decreasing degrees of abstraction. The full arrows indicate relationships that have already been formalized to a certain extent, the dashed ones indicate relationships that require considerable additional work before they are understood and formalized to the same extent.

Hence, as the title indicates, the material in this paper is still in an evolutionary state (not yet fully "digested") and contains many unresolved issues. As a consequence, this presentation can only indicate a general direction, not provide conclusive answers.



## 1. REPRESENTATIONAL AND DENOTATIONAL SEMANTICS

### 1.0 Basic paradigm and its variants

#### a. Representation, denotation, realization

As outlined in [BOUT86], we want to relate an "abstract" operation  $f : X \rightarrow Y$  to its "concrete" realization. The quote marks indicate that the terms "abstract" and "concrete" are only used as a paradigm, not in some strict sense. Corresponding to the domains  $X$  and  $Y$ , in this context called denotation domains, we define so-called representation domains  $X'$  and  $Y'$  together with two mappings

the representation mapping  $r : X \rightarrow X'$

the denotation mapping  $d : Y' \rightarrow Y$

We say that a "concrete" operation  $c_f : X' \rightarrow Y'$  realizes  $f$  (with respect to  $d$  and  $r$ )

if  $f = d \circ c_f \circ r$  (where  $\circ$  denotes function composition).

#### b. Variants

- If  $X = Y$  then we impose the condition that  $X' = Y'$  and  $d \circ r = id_Y$ .
- If  $X = Y \times Y$  then  $X' = Y' \times Y'$  and  $r : Y \rightarrow Y'$  with  $d \circ r = id_Y$   
and  $c_f$  realizes  $f$   $\iff \exists (y, y') : X. (d \circ c_f(r y, r y') = f(y, y'))$ .

- $r$  and  $c_f$  need not be "proper" (i.e. single-valued) functions but may be arbitrary relations. Yet we keep using the notation for function and for application because this is more convenient than the traditional formulation using powersets. This issue is not further discussed in this overview.

### c. Applications

- In the area of digital systems:
  - . Data processing (see examples in subsections 1.1 and 1.2).
  - . Data transmission:  $f$  is the identity function on  $X$ ,  $c_f$  represents the behaviour of an unreliable channel,  $r$  and  $d$  are the encoding and decoding functions for an error-correcting code.
  - . Data encryption:  $f$  is the identity function on  $X$ ,  $c_f$  the identity function on  $X'$ ,  $r$  and  $d$  are the encoding and decoding functions for a cipher.
- In the area of analog systems:
  - . Complex representation of sinusoidal functions (one-phase, multi-phase, symmetric components).
  - . Transforms (Fourier, Laplace, Hilbert etc.)

### 1.1 Application to combinational functions

As an example, we describe two's complement number arithmetic.

- Assume that unsigned integer arithmetic has been defined (exercise), i.e.

$$\mathbb{N}_n = 0..(2^n - 1) \quad B = \{0, 1\}$$

$$du_n : \mathbb{N}_n \rightarrow B^n \quad ru_n : B^n \rightarrow \mathbb{N}_n \quad ru = du^{-1}$$

$$adu_n : B^n \times B^n \rightarrow B^{n+1} \text{ realizing } + : \mathbb{N}_n \times \mathbb{N}_n \rightarrow \mathbb{N}_{n+1}$$

- Define two's complement representation as follows:

$$\mathbb{Z}_n = (-2^{n-1})..(2^{n-1}-1) \quad B = \{0, 1\}$$

$$rs_n : \mathbb{Z}_n \rightarrow B^n \text{ with } rs_n m = ru_n ( \text{mod } 2^n m )$$

$$ds_n : B^n \rightarrow \mathbb{Z}_n \text{ with } ds_n = rs_n^{-1}$$

- Define the operation  $ads_n = rad_n \circ adu_n$

$$\text{where } rad_n : B^{n+1} \rightarrow B^n \text{ with } rad_n(a_n \dots a_0) = a_{n-1} \dots a_0$$

Property:  $rad_n$  realizes mod  $2^n$  (w.r.t.  $ru_n$  and  $du_n$ ).

Main theorem:  $ads$  realizes  $+$  over  $P_n$  (w.r.t.  $rs_n$  and  $ds_n$ )

where  $P_n = \{(m, m') : \mathbf{Z}_n^2 \mid (m + m') : \mathbf{Z}_n\}$

Proof (illustrating the transformational style). Let  $(m, m') : P_n$ .

$$\begin{aligned}
 ds_n \circ ads_n (rs_n m, rs_n m') &= \{\text{def } ads_n\} ds_n \circ rad_n \circ adu_n (rs_n m, rs_n m') \\
 &= \{\text{def } rs_n\} ds_n \circ rad_n \circ adu_n (ru_n (\text{mod } 2^n m), ru_n (\text{mod } 2^n m')) \\
 &= \{adu_n \text{ realizes } +\} ds_n \circ rad_n \circ ru_{n+1} (\text{mod } 2^n m + \text{mod } 2^n m') \\
 &= \{\text{rad realizes mod } 2^n\} ds_n \circ ru_n (\text{mod } 2^n (\text{mod } 2^n m + \text{mod } 2^n m')) \\
 &= \{\text{property mod}\} ds_n \circ ru_n (\text{mod } 2^n (m + m')) \\
 &= \{\text{def } rs_n \text{ and } (m, m') : P_n\} ds_n \circ rs_n (m + m') \\
 &= \{ds_n = rs_n^{-1}\} m + m'
 \end{aligned}$$

(end of theorem)

Notice that this proof exactly captures the essence of two's complement arithmetic.

## 1.2 Application to sequential functions

### a. Sequential functions and the concept of state

- a function  $f : I^* \rightarrow U^*$  is called sequential if  
 $x \leq y \Rightarrow f x \leq f y$  ( $\leq$  means "is prefix of").
- The set of all sequential functions from  $I^*$  to  $U^*$  is written  $S(I, U)$ .
- Property:  $f : I^* \rightarrow U^*$  is sequential iff there exists a function  $f' : I^* \rightarrow U^*$  such that, for all  $x$  in  $I^*$  and all  $i$  in  $I^1$ ,  $f(x.i) = (fx).f'(x.i)$ . This function  $f'$  (which can be shown to be unique if it exists) is called the derivative of  $f$ .
- Given  $f : S(I, U)$ , the state generator function is defined as  
 $\hat{f} : I^* \rightarrow I^* \rightarrow U^*$  with  $\hat{f} x y = f'(x.y)$
- The state set of  $f$  is defined as  $Q_f = \{\hat{f} x \mid x : I^*\}$   
 Note that  $Q_f$  is a subset of  $I^* \rightarrow U^*$ .  $\hat{f} x$  is called the state after  $x$ .

### b. Generalized sequential machines

We define the set of generalized sequential machines over  $I, U, Q$  as

$$M(I, U, Q) = (Q \rightarrow I \rightarrow U^*) \times (Q \rightarrow I \rightarrow Q) \times U^* \times Q$$

For a typical instance  $m : M(I, U, Q)$  the elements are

$$m = (\text{output function, next-state function, prelude, initial state})$$

## c. Representational and denotational semantics of sequential machines

Denotation function ("behaviour")

```

|| behav : M(I, U, Q) → S(I, U)
def behav m x = prel++(aux start x)
      where || aux : Q → S(I, U) with
            aux q () = ()
            aux q (i:x) = (out q i)++(aux (next q i) x)
            (out, next, prel, start) = m

```

Representation function ("realization")

```

|| real : S(I, U) → M(I, U, I* → U*)
def real f = (out, next, prel, start)
      where out q i = q i,
            next q i x = q(i:x)
            prel = f()
            start = deriv f

```

Remarks: for these definitions we have used the language SASL [TURN75], hence the double vertical bars || for comments, the symbol ++ (instead of the dot .) for concatenation, and the notation i:x for prefixing i to the sequence x. The definition of the derivative function is left as an exercise.

## 2. DESCRIPTION OF DIGITAL SYSTEM ARCHITECTURES

2.0 Separation of architectural concernsa. View of the machine

A computer is viewed as a collection of

- . individually named registers (ACC, PC, index reg)
- . adressable locations (memory, I/O)

jointly constituting the system state. This is described by means of a state type definition, for instance

```

Sys = Proc × Stor × Out where
  Proc = Regs × Pcount × Index × Cond where
    Regs = (0..k-1) → {0, 1}n
    Pcount = Index = {0, 1}m
    Cond = C × V × Z × S where C, V, Z, S = {0, 1}
  Stor = (0..2m-1) → {0, 1}n

```

$$\text{Out} = (0..p-1) \rightarrow \{0, 1\}^n$$

This type definition contains the same information (and hence can conveniently replace) the PMS notation devised by Bell and Newell [SIEW82]

#### b. View of the program

A program is viewed as a sequence of instructions. Every instruction specifies (in encoded form):

- . one or more concrete operations (discussed in section 1)
- . origin (source) of the arguments of the operation(s)
- . destination(s) of the result(s) of the operation.

The retrieval of this information is described by an "(instruction) decoding function". Note that in such a description all affected destinations are explicit, hence there is no more distinction between "primary effects" and "side effects".

#### c. View of the combination program/machine

An "effect function" specifies the state transformation resulting from the execution of an instruction:

effect : Instr  $\rightarrow$  Input  $\rightarrow$  Sys  $\rightarrow$  Sys

At first sight, this style of description is somewhat reminiscent of denotational semantics [STOY77, GORD79], but in fact it is essentially different since it reflects the structure of the machine and therefore is very operational in nature.

These various points are illustrated by means of two cross-sections: one complete description of a simple (but fully representative) architecture (subsection 2.1) and one partial description of a typical issue in more complex architectures (subsection 2.2).

### 2.1 Example: Binary Decision Machine (BDM)

#### a. General characteristics

The Binary Decision Machine [BOUT76] is a microprogrammable device with  $n$  binary inputs and  $n$  binary outputs. It is used as a programmable controller replacing hardwired logic in various applications [PIGU81]. Despite its very simple architecture, we shall see that it contains all essential elements of a computer.

## b. Instruction format

$$\text{Instr} = C \times V \times S \times A$$

where  $C = \{0, 1\}$  ("opcode")       $S = 0..(n-1)$  ("select")

$V = \{0, 1\}$  ("value")       $A = 0..(N-1)$  ("address")

c. Informal description of the effect of  $\text{instr} = (c, v, s, b)$ 

If  $c = 0$  ("branch") then  $v$  is compared to  $\text{input}(s)$ .

If these values are equal, then branch to address  $b$ .

If  $c = 1$  ("output") then  $\text{output}(s)$  is set to value  $v$ .

## d. Formal functional description of the BDM

## - Effect of an instruction:

Input and output spaces  $I = S \rightarrow \{0,1\}$  and  $U = S \rightarrow \{0,1\}$

State space  $\text{Sys} = A \times U$

$\text{effect} : \text{Instr} \rightarrow I \rightarrow \text{Sys} \rightarrow \text{Sys}$

$\text{effect}(c, v, s, b) i(a, u) = (c = 1) \rightarrow (b, \text{assign } s \ v \ u);$

$i \ s = v \rightarrow (b, u); (a + 1, u)$

The auxiliary definition "assign" is a typical example of the functional description of a partial state change:

$\text{assign} : S \rightarrow V \rightarrow U \rightarrow U$  (recall  $U = S \rightarrow \{0, 1\}$ )

$\text{assign } s \ v \ u \ s' = (s' = s) \rightarrow v ; u \ s'$

## - Composition of the machine with the program:

Program space  $P = A \rightarrow \text{Instr}$

$\text{bdm} : P \rightarrow M'(I, U, \text{Sys})$  (ordinary sequential machines)

$\text{bdm } p = (\text{out}, \text{next}, \text{start})$

where  $\text{out}(a, u) i = u$

$\text{next}(a, u) i = \text{effect}(p \ a) i(a, u)$

$\text{start}(a, u) = (0, \text{zero})$  where  $\text{zero } s = 0$

## - Behaviour of the machine with the program:

$\text{behav}'(\text{bdm } p) : I^* \rightarrow U^*$

where  $\text{behav}'$  is a simplification of the definition given in section 1.2 to cover ordinary sequential machines.

2.2 Example: addressing modes

Assume that the instruction decoding function has yielded "references" for the sources and destinations. For our purpose, references are characterized by three functions that can be applied to them:

$\text{mode} : \text{Ref} \rightarrow \text{Amode}$

```
data : Ref → Adata
size : Ref → Dsize [optional]
```

For instance

```
Ref = Amode × Adata
Amode = {V, D, I} [Value, Direct, Indirect]
Adata = {0, 1}n {0, 1}m [words, addresses]
```

where the functions "mode" and "data" are just projection functions selecting components of a Cartesian product. In general, two meaning functions will be needed for references:

. For references indicating sources only (returning values)

```
mS : Ref → Sys → Val
mS r sys = mode r = V → d;
           mode r = D → deref d;
           mode r = I → deref (deref d)
           where d = data r
           deref d = cont (stor d) sys
```

. for references indicating destinations as well

```
mD : Ref → Sys → Sel [abstract selector]
mD r sys = || mode r = V does not occur
           mode r = D → stor d
           mode r = I → stor (deref d)
           where d = data r
           deref d = cont (stor d) sys
```

Without going into too much detail, let us clarify the auxiliary functions:

- stor d yields a composite field selector ("abstract selector") to the d-th storage location in an element of type Sys.
- cont s sys yields the contents of the field indicated by the selector s in sys.

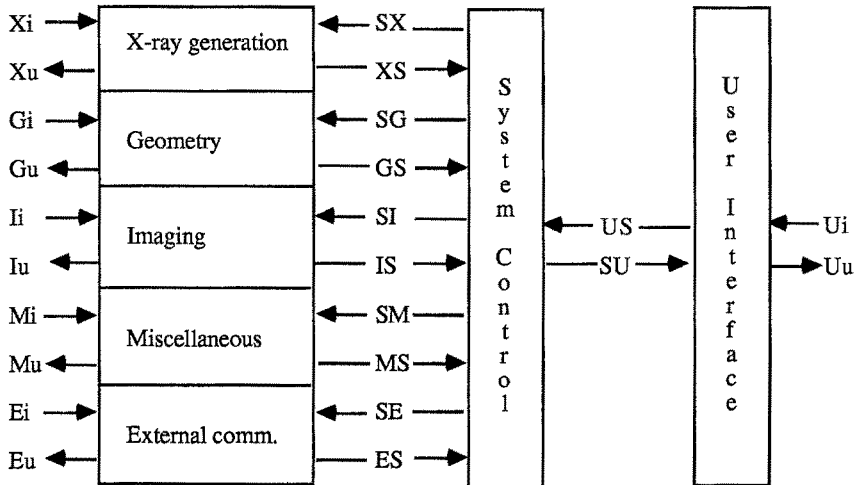
### 3. FUNCTIONAL DESCRIPTION OF "MIXED" SYSTEMS

#### 3.0 Functional description of system composition

##### a. Example: X-ray system

The following example is taken from a student project in collaboration with a manufacturer of medical electronic equipment. Consider an X-ray system whose control part is pictorially (and informally) represented by the diagram below.

One must imagine this diagram extended to the left by the controlled physical system (the X-ray equipment) and to the right by the user (the X-ray technician). The X-ray equipment contains digital subsystems and analog subsystems.



#### b. Desired functional interface description

We would like to reflect system structure by a formal definition of the following form:

```

sys : (Ui × Xi × Gi × ... × Ei) → (Uu × Xu × Gu × ... × Eu)
sys (ui, xi, gi, ..., ei) = (uu, xu, gu, ..., eu)
  where (us, uu) = usint (su, ui)
        (sx, sg, ..., se, su) = sysctrl (xs, gs, ..., es, us)
        (xs, xu) = xraygen (xi, sx)
        ...
        (es, eu) = externcomm (ei, se)
  
```

Whereas the 1-1 correspondence with the picture (i.e. the structural interpretation) is very clear, for the behavioural interpretation the "variables" xi, ui etc. and the "functions" usint, sysctrl etc. must also be given some useful meaning, making it possible to describe the behaviour mathematically.

### 3.1 Modelling problems

#### a. Existing "languages" for general system description

Some languages allow to describe subsystems and interconnections

- . by boxes and arrows, as in SADT [ROSS77]
- . by textual statements, as in PSL/PSA [TEIC77]

but they do not provide a formalism to attach meaning to the symbols (boxes, arrows, names), except in the form of a comment, and have no useful formal properties.

#### b. Issues to be dealt with

We wish to ensure continuity with the currently used mathematical models for "homogeneous" systems, viz.

- Digital systems : the state machine model.  
Semantic domain : sequences of type  $A^*$
- Analog systems : discrete-time (sampled data) system models [FRAN80].  
Semantic domain : discrete-time functions of type  $T \rightarrow A$   
where  $T = \mathbb{N}$  or  $T = \mathbb{Z}$

In the following two subsections, we consider two main issues in this respect:

- Definition of a uniform semantic domain (3.2)
- Cooperation of discrete-time systems with different sampling periods (3.3)

### 3.2 Semantic domains for mixed systems

#### a. Unification of sequences and discrete-time functions

| operation  | discrete-time functions                                                                                                                               | sequences                                                                   |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------|
|            | -----                                                                                                                                                 | -----                                                                       |
| prefixing  | $\text{pr} : A \rightarrow (\mathbb{N} \rightarrow A) \rightarrow (\mathbb{N} \rightarrow A)$<br>$\text{pr } a \ f \ n = n = 0 \rightarrow a; f(n-1)$ | $\text{pr} : A \rightarrow A^* \rightarrow A^*$<br>$\text{pr } a \ x = a:x$ |
| head       | $\text{hd} : (\mathbb{N} \rightarrow A) \rightarrow A$<br>$\text{hd } f = f \ 0$                                                                      | $\text{hd} : A^+ \rightarrow A^*$<br>$\text{hd } (a:x) = a$                 |
| tail       | $\text{tl} : (\mathbb{N} \rightarrow A) \rightarrow (\mathbb{N} \rightarrow A)$<br>$\text{tl } f \ n = f(n+1)$                                        | $\text{tl} : A^+ \rightarrow A^*$<br>$\text{tl } (a:x) = x$                 |
| (property) | $\text{pr } (\text{hd } f) \ (\text{tl } f) = f$                                                                                                      | $\text{pr } (\text{hd } x) \ (\text{tl } x) = x$                            |

To maintain the distinction, we shall write  $\text{spr}$ ,  $\text{shd}$ ,  $\text{stl}$  for the operations involving sequences.

## b. Relation between the two formulations

- Compatibility relation  $C : A^* \times (N \rightarrow A)$   
defined by  $\epsilon \in C$  for empty sequences  

$$x \in C f = (\text{shd } x = \text{hd } f) \wedge ((\text{stl } x) \in C (\text{tl } f))$$
- Similarity. Let  $E(I, U) = \{f : S(I, U) \mid \text{length}(fx) = \text{length } x\}$ .  
The sequential functions  

$$\text{st} : E(I, U) \quad \text{and} \quad \text{ft} : (N \rightarrow I) \rightarrow (N \rightarrow U)$$
are called similar if  $s \in C f \Rightarrow (\text{st } s) \in C (\text{ft } f)$ . The idea is that similar sequential functions describe the same system behaviour.

## c. Machine behaviour (for ordinary SM's)

For the set of ordinary sequential machines

$$M(I, U, Q) = (Q \rightarrow I \rightarrow U) \times (Q \rightarrow I \rightarrow Q) \times Q$$

we define two behaviour functions

$$|| \text{ GF} : M(I, U, Q) \rightarrow Q \rightarrow (N \rightarrow I) \rightarrow (N \rightarrow U)$$

def  $\text{GF } m \text{ q } f = \text{pr } (\text{out } q (\text{hd } f)) (\text{GF } m (\text{next } q (\text{hd } f)) (\text{tl } f))$

where  $(\text{out}, \text{next}, \text{start}) = m$

$$|| \text{ GS} : M(I, U, Q) \rightarrow Q \rightarrow E(I, U)$$

def  $\text{GS } m \text{ q } x = (x = \epsilon) \rightarrow \epsilon;$

$\text{spr } (\text{out } q (\text{shd } x)) (\text{GS } m (\text{next } q (\text{shd } x)) (\text{stl } x))$

where  $(\text{out}, \text{next}, \text{start}) = m$

Theorem:

For all  $q$  in  $Q$ ,  $\text{GS } m \text{ q}$  and  $\text{GF } m \text{ q}$  are similar. (Proof: exercise)

Having expressed sequential machine behaviour in terms of discrete-time functions, we have established the correspondence with the formalism used for sampled-data systems.

### 3.3 Timing and scheduling aspects

Consider a discrete-time system in which subsystems have different sampling periods or "activation times". There are two main approaches to a solution:

- a separate operating system (functional style, of course). This solution may be the only one for highly unstructured systems. The disadvantage is that the relation between the subsystems is hidden inside the scheduler.
- Associating scheduling directly with the scheduled subsystem (localized). This is the solution proposed next.

## b. Scheduled sequential machines

The following solution assumes that the sampling periods are commensurable and that the largest common measure is taken as the time unit. The scheduling period (in number of time units) is supplied to a "scheduled behaviour function" as an extra parameter>

```

|| bsched : N → M(I, U, Q) → (N → I) → (N → U)
def bsched per m = G start
where || G : Q → (N → I) → (N → U)
      G q i n =
        (n = 0) → out q (i 0);
        mod per n = 0 → G (next q (i 0)) (shift per i) (n - per);
        G q i ((div per n) * per)
      shift per i n = i(n + per)
      (out, next, start) = m

```

## 4. THE FUNCTIONAL TEMPORAL CALCULUS

### 4.1 Motivation - Basic approach

#### a. Purpose of the formalism

We are concerned with the analysis of time-varying phenomena in

- digital systems (hardware, software, synchronous, asynchronous),
- analog systems (electronic, mechanic, etc.).

#### b. Present state of the art

With respect to the usage of mathematical models, there exists a divided situation:

- In applied mathematics and engineering,
  - . the formalism is based on calculus and analysis [PAP084],
  - . "time hiding" is achieved by operators and transforms [BRAC78],
  - . (well-proven) application areas are: electronics, control [FRAN86], communication [CARL74], stochastic processes [PAP065] etc.
- In (theoretical) computer science,
  - . the formal system, viz. temporal logic, is based on logic and model theory [MANN83],
  - . "time hiding" is achieved by axiomatization [MANN82, 83],

- . (tentative) application areas are: digital circuits [BOCH82, CLAR83, BROW84], concurrency [MANN82, KARP84] network protocols [SCHW82] etc.

#### c. Problem with this situation

The formal approach of temporal logic prohibits linking to other formalisms (in particular the formalisms based on calculus and analysis used in applied science and engineering). This "closed semantics" is typical for the purely axiomatic approach where the model is "hidden".

#### d. Proposed approach

The "Functional Temporal Calculus" unifies all aforementioned calculi in a single framework with "open semantics" (i.e. the model is part of the formalism, axioms become theorems). The following exposition deals only with this unification; the reader who finds this material somewhat abstract because application examples are missing is encouraged to study some of the referenced material [MANN82, BOCH82] beforehand.

### 4.2 Basic definition - Concept of state

#### a. Temporal variables

Time-varying phenomena are characterized by

- . a static value domain or static type, that is: the domain of instantaneous values
- . temporal variables, defined as functions

$$y : T \times \Sigma \rightarrow V_y$$

where  $T$  is the time domain,  $V_y$  is the static type of  $y$ , and  $\Sigma$  is the set of possible outcomes (the role of  $\Sigma$  will gradually become clear in the sequel).

We use the following notation for function application:

- .  $T$  continuous:  $y(t)\sigma$ , with Currying :  $y(t)$  and  $y\sigma$
- .  $T$  discrete:  $y_n\sigma$ , with Currying :  $y_n$  and  $y\sigma$

This notational distinction is a concession to existing practice, but may well disappear in future formulations because it serves no useful purpose.

## b. Concept of state

We define the state  $s$  as the joint behaviour of all temporal variables, viz.

$$s = \langle x, \dots, z \rangle \quad (\text{hence the type is } T \times \Sigma \rightarrow V_x \times \dots \times V_z)$$

with the convention that application distributes over  $\langle \rangle$ , e.g.

$$s\sigma = \langle x\sigma, \dots, z\sigma \rangle$$

For certain theorems of FTC, the following explicit definition is assumed for  $\Sigma$ :

$$\Sigma = T \rightarrow V_x \times \dots \times V_z$$

with the condition that the state  $s$  satisfies  $s(t)\sigma = \sigma t$ . In that case, we define two modifiers on  $\sigma$ :

- Time translation : given  $\sigma: \Sigma$  and  $t: T$ , then

$$\sigma(t): \Sigma \quad \text{with} \quad \sigma(t)t' = \sigma(t+t')$$

- Variable replacement: given  $\sigma: \Sigma$ , a variable name  $y$  and a replacement  $b: T \times \Sigma \rightarrow V_y$ , then

$$\sigma_{[y \rightarrow b]}: \Sigma \quad \text{with} \quad \sigma_{[y \rightarrow b]}t = \langle x(t)\sigma, \dots, b(t)\sigma, \dots, z(t)\sigma \rangle$$

where  $b(t)\sigma$  appears in the position corresponding to  $y$ .

## 4.3 Temporal domains, mappings and combinators

### a. Temporal domains

With every (static) domain  $D$  we associate a temporal domain  $T \times \Sigma \rightarrow D$ . The letter  $W$  henceforth designates the (static) truth domain {true, false}.

### b. The Functional Temporal Calculus (FTC)

FTC is the calculus of functions between temporal domains, i.e. functions of type

$$(T \times \Sigma \rightarrow D) \rightarrow (T \times \Sigma \rightarrow D').$$

We distinguish between two kinds of such functions:

- Temporal mappings that can be derived from static mappings by extension, viz. given  $f: D \rightarrow D'$ , we define

$$f': (T \times \Sigma \rightarrow D) \rightarrow (T \times \Sigma \rightarrow D') \quad \text{with} \quad (f'd)(t)\sigma = f(d(t)\sigma)$$

This kind of extension is quite common in applied science and engineering. It is also customary to omit the apostroph since the distinction is always clear from the context. Static constants are viewed as functions without argument, hence their extensions are constant-valued temporal variables.

- Temporal combinators cannot be derived in this way but require individual definitions. Typical combinators are:

- The shift combinator, for instance (in discrete  $T$ ):

$$O: (T \times \Sigma \rightarrow D) \rightarrow (T \times \Sigma \rightarrow D)$$

$$(Od)_n \sigma = d_{n+1} \sigma$$

- The last combinator, for instance (in continuous T):  

$$LI : (T \times \Sigma \rightarrow W) \rightarrow (T \times \Sigma \rightarrow T \quad \{-\infty\}) \quad (I \text{ stands for "inclusive"})$$

$$(LI \ w)(t)\sigma = \underline{As} \leq t. \sim w(t)\sigma \rightarrow -\infty;$$

$$s \text{ satisfying } s \leq t \wedge w(s)\sigma \wedge \underline{Ar} : (s, t]. \sim w(r)\sigma$$
- The let clause (parameterized by a variable name  $y$  and a replacement  $b : T \times \Sigma \rightarrow V_y$ ):  

$$\underline{\text{let}} \ y = b \ \underline{\text{in}} : (T \times \Sigma \rightarrow D) \rightarrow (T \times \Sigma \rightarrow D)$$

$$(\underline{\text{let}} \ y = b \ \underline{\text{in}} \ d)(t)\sigma = d(t)\sigma_{[y \leftarrow b]}$$

#### 4.4 Extensions and relation to other formalisms

##### a. The logic subset of Functional Temporal Calculus

As shown in [BOUT85], the following subset of FTC is sufficient to subsume temporal logic [MANN83] and provides some generalization in the process. We assume that  $T = \mathbb{Z}$  (discrete time domain).

- The temporal mappings of this subset are the extensions of all static functions (between value domains), static predicates (from value domains to truth domains) and logical operators (between truth domains).
- The temporal combinators are, in addition to the shift operator defined earlier, the following four combinators of type  $(T \times \Sigma \rightarrow W) \rightarrow (T \times \Sigma \rightarrow W)$ :

$$\begin{aligned} \Box \text{ ("always")} & \quad (\Box w)_n \sigma = \underline{Ai} \geq n. w_i \sigma \\ \Diamond \text{ ("sometime")} & \quad (\Diamond w)_n \sigma = \underline{Ei} \geq n. w_i \sigma \\ \forall y. \text{ ("for all y")} & \quad (\forall y. w)_n \sigma = \underline{Ab} : V_y. w_n \sigma_{[y \leftarrow b]} \\ \exists y. \text{ ("for some y")} & \quad (\exists y. w)_n \sigma = \underline{Eb} : V_y. w_n \sigma_{[y \leftarrow b]} \end{aligned}$$

- The Frege-operator establishing the reference to the time origin

$$\begin{aligned} |- : P(\Sigma) \times (T \times \Sigma \rightarrow W) & \rightarrow W \quad \text{with} \\ C|-w = \underline{A}\sigma : C.w_0 \sigma & \quad (\text{infix notation}) \end{aligned}$$

By convention,  $|- w$  denotes  $\Sigma \mid - w$

##### b. Relation to stochastic processes

Following [PAP065], we define a stochastic process as a function

$$y : T \times \Sigma \rightarrow V_y$$

assigning to every outcome  $\sigma$  in  $\Sigma$  a time function  $y_\sigma : T \rightarrow V_y$ . For certain subsets of  $\Sigma$ , called events, probabilities are defined. All definitions from [PAP065] now carry over directly. For instance, for a real-valued temporal variable  $y$  ( $V_y = \mathbb{R}$ ),

the probability density function  $F_y$  is defined as follows:

$$F_y : V_y \times T \rightarrow [0, 1]$$

$$F_y(\eta, t) = P\{\sigma : \Sigma \mid y(t)\sigma \leq \eta\}$$

where  $P e$  denotes the probability of event  $e$ .

### c. Continuous-time Functional Temporal Calculus

The principles have already been discussed in sections 4.1 and 4.2 (assuming  $T = \mathbb{R}$ ). Let us only show how some other calculi, for instance the one proposed in [AMBL85], can be subsumed. If we define the following auxiliary functions

- Temporal function composition

$$_o : (T \times \Sigma \rightarrow D) \times (T \times \Sigma \rightarrow T) \rightarrow (T \times \Sigma \rightarrow D) \quad \text{with}$$

$$(d \circ k)(t)\sigma = d(k(t)\sigma)\sigma$$

- Event (in the spirit of [AMBL85])

$$\uparrow : (T \times \Sigma \rightarrow W) \rightarrow (T \times \Sigma \rightarrow W)$$

$$(\uparrow w)(t)\sigma = \underline{Es} > 0. (Ar:(t-s, t). \sim w(r)\sigma \wedge Ar:(t, t+s). w(r)\sigma)$$

For instance, the behaviour of a d-type flipflop with data input  $d$ , clock input  $c$  and output  $q$  can now be expressed as

$$q = d \circ (LE \uparrow c)$$

In words: " $q$  has the value that was present at  $d$  at the time instant of the last rising edge of  $c$ ." The function  $LE$  ( $E$  = "exclusive") is similar to  $LI$  but restricted to zero-duration events (such as  $c$ ).

### d. Relation to system theory

A dynamical system is characterized by a function

$$f : (T \rightarrow E) \rightarrow (T \rightarrow U) \quad [\text{FRAN80}]$$

which can be seen as "simplified" versions of

$$f : (T \times \Sigma \rightarrow E) \rightarrow (T \times \Sigma \rightarrow U)$$

Indeed,  $\Sigma$  is important in stochastic system theory only. Let us exemplify the correspondence between FTC and discrete-time systems ( $T = \mathbb{Z}$ ). It is a simple exercise [BOUT85] to show that the  $z$ -transform operator [FRAN80]

$$Z : (\mathbb{Z} \rightarrow E) \rightarrow E[z^{-1}] \quad (\text{formal polynomials in } z^{-1} \text{ over } E)$$

$$Ze = \sum_{k=-\infty}^{+\infty} e_k \cdot z^{-k}$$

and the input/output transfer "function"  $H(z)$  [FRAN80] are related to the shift-operator as follows:

$$Zu = H(z).Ze$$

$$u = H(0)e \quad \text{where } H(0) = (\lambda z.H(z))0$$

Here  $.$  denotes multiplication, whereas juxtaposition denotes application.

#### 4.5 Reasoning in the logic subset of FTC

##### a. Purpose of this subsection

The main point is to convince the reader that the logic subset of FTC indeed subsumes temporal logic, in particular that all axioms and inference rules of [MANN83] become theorems of FTC. Only a few examples are given here; the complete treatment can be found in [BOUT85].

##### b. Example - for "axiom A3" in [MANN83]

Theorem A3 :  $\vdash \Box w \Rightarrow w$

Proof  $\vdash \Box w \Rightarrow w$

$$= \{\text{def } \vdash\} \quad \underline{A}\sigma : \Sigma. (\Box w \Rightarrow w)_{0\sigma}$$

$$= (\text{extension } \Rightarrow) \quad \underline{A}\sigma : \Sigma. [(\Box w)_{0\sigma} \Rightarrow w_{0\sigma}]$$

$$= \{\text{def } \Box\} \quad \underline{A}\sigma : \Sigma. (\underline{A}1 \geq 0. w_{1\sigma} \Rightarrow w_{0\sigma})$$

$$= \{\text{elementary logic}\} \text{ true}$$

q.e.d.

##### c. Remark

Axioms and inference rules of temporal logic are specified in syntactic terms. Hence in FTC some concepts from temporal logic have to be redefined in semantic terms. This leads to several interesting generalizations because the syntactic criteria of temporal logic are sufficient, but usually not necessary conditions for the semantic criteria of FTC to be satisfied. Example:

FTC definition: if  $w : T \times \Sigma \rightarrow W$  then  $b : T \times \Sigma \rightarrow V_y$  is said to be - globally free for y in w if, for all  $(n, \sigma)$  in  $T \times \Sigma$ ,  $w_n^\sigma[y \leftarrow b] = w_n^\sigma[y \leftarrow b_n \sigma]$

FTC theorem (subsumes axiom A12 in [MANN83]):

If b is globally free for y in w then

$$\vdash \forall y. w \Rightarrow (\text{let } y = b \text{ in } w)$$

#### 5. CONCLUSIONS

Issues not discussed in this paper but satisfactorily resolved:

- Functional recast of the theory of digital systems
- Rapid simulator prototyping of digital circuit descriptions
- Complete proof style in FTC subsuming temporal logic

Issues not discussed but (only) partially resolved:

- Stylish description of data flow in "messy" computer architectures (i.e. with many side-effects and exceptions),
- Unified framework for "mixed" systems where not all subsystems can be viewed as discrete-time systems,
- Subsuming other variants of temporal logic in FTC (e.g. "branching-time logic").

Main unresolved issue:

- How to close the gap between specification and implementation descriptions at higher levels in a satisfactory way.

In the literature one finds many formalisms with the claim that they resolve latter issue, but the validity of such a claim obviously depends on the standards one sets for "satisfactory". The usefulness such formalisms might have for engineering purposes is greatly diminished by the overhead resulting from issues at the meta-level that are of interest to logicians only. Much work remains to be done to recast some of these results in the framework of applied mathematics.

#### REFERENCES

- [AMBL85] P. Amblard et al., "Describing and reasoning about circuits behaviour by means of time functions". Proc. 1975 on Computer Hardware Description Languages, pp. 39-48, Tokyo (Aug. 1985)
- [BOCH82] G.V. Bochmann, "Hardware specification with temporal logic: an example". IEEE trans. C-31, 3 pp. 223-231 (March 1982)
- [BOUT76] R.T. Boute, "The Binary Decision Machine as a programmable controller". Euromicro Newsletter 2, 1, pp. 16-22 (Jan. 1976)
- [BOUT85] R.T. Boute, "Functional temporal calculus versus temporal logic: an engineering approach". Report No. 59, Dept of Computer Science, University of Nijmegen, (March 1985)
- [BOUT86] R.T. Boute, "Elements for the formal description of systems ". In: A. Kündig, ed., New Approaches to the Formal Description and the Design of Embedded Systems. (March 1986)
- [BRAC78] R.N. Bracewell, The Fourier transform and its applications, 2nd ed. McGraw-Hill, New York (1978)
- [BROW84] M. Browne et al., "Automatic verification of sequential circuits using temporal logic". Report CMU-CS-85-100, Dept. of Computer Science, Carnegie-Mellon University (Dec. 1984)
- [CARL74] A.B. Carlson, Communications systems (2nd. ed.), McGraw-Hill (1974)
- [CLAR83] E.M. Clarke et al., "Automatic verification of finite-state concurrent systems using temporal logic specifications; a practical approach. Report CMU-CS-83-152, Dept. of Computer Science, Carnegie-Mellon University (Sept. 1983)
- [FRAN80] G.F. Franklin, J.D. Powell, Digital control of dynamic systems. Addison-Wesley, Reading, Mass. (1980)
- [FRAN86] G.F. Franklin, J.D. Powell, A. Emami-Naeini, Feedback control of dynamic systems. Addison-Wesley, Reading, Mass. (1986)
- [GORD79] M.J.C. Gordon, The denotational description of programming languages. Springer, Berlin (1979)
- [KARP84] R.A. Karp, "Proving failure-free properties of concurrent systems using temporal logic", ACM Trans. on Prog. Lang. and Systems 6, 2, pp. 239-253. (April 1984)

- [MANN82] Z. Manna, A. Pnueli, "Verification of concurrent programs: the temporal framework". In: R.S. Boyer, J.S. Moore, eds., The correctness problem in Computer Science, pp. 215-273. International Lecture Series in Computer Science, Academic Press, London (1982)
- [MANN83] Z. Manna, A. Pnueli, "Verification of concurrent programs: a temporal proof system". Report No. STAN-CS-83-976. Dept. of Computer Science, Stanford University, Stanford, Ca. (June 1983)
- [PAPO65] A. Papoulis, Probability, random variables and stochastic processes. McGraw-Hill, New York (1965)
- [PAPO84] A. Papoulis, Signal Analysis. McGraw-Hill, New York (1984)
- [PIGU81] C. Piguet et al., "Microcomputer design for digital watches". Proc. 7th Euromicro Symposium, pp. 431-442 (Sept. 1981)
- [ROSS77] D.T. Ross, "Structured Analysis (SA): a language for communicating ideas". IEEE SE-3, 1, pp. 16-34 (Jan. 1977)
- [SCHW82] R.L. Schwartz, P.M. Melliar-Smith, "From state machines to temporal logic: specification methodes for protocol standards". IEEE Trans. COM-30, 12, pp. 2486-2496 (Dec. 1982)
- [SIEW82] D.P. Siewiorek, C.G. Bell, A. Newell, Computer structures: principles and examples. McGraw-Hill, New York (1982)
- [STOY77] J.E. Stoy, Denotational semantics: the Scott-Strachey approach to programming language theory. MIT Press, Cambride, Mass. (1977)
- [TEIC77] D. Teichroew, E.A. Hershey III, "PSL/PSA: a computer-aided technique for structured documentation and analysis of information processing systems". IEEE SE-3, 1, pp. 41-48 (Jan. 1977)
- [TURN79] D.A. Turner, SASL language manual (revised version). University of Kent (1979)

## TOOLS FOR PROTOCOLS DRIVEN BY FORMAL SPECIFICATIONS

Harry Rudin  
IBM Research Division, Zurich Research Laboratory  
8803 Rüschlikon, Switzerland  
Telephone- 01 - 724 84 33  
Telex- 826 561 ibmr  
Network- hr @ ibm.com

Formal methods are being used increasingly for computer-communication protocols. The use of a formal description promotes conciseness and avoids dangerous ambiguity. Once a formal description is available, particularly in executable form, partly automated techniques can be used for: 1) examining the protocol specification for certain kinds of errors; 2) estimating its performance; 3) implementing the protocol as part of a product; 4) testing the implementation for architectural conformance; and 5) designing a conversion mechanism to be the bridge between different protocol systems. While there are alternative formal description approaches, this paper concentrates on the formal approach most used, that based on a finite-state-machine definition.

### I. INTRODUCTION

With the passing of time, information-handling systems now handle increasing function and have become increasingly coupled with one another. This coupling among the various, often distributed parts is achieved via "protocols". The complexity of even a stand-alone information system is now so great that it must be partitioned into modules of comprehensible size, creating the necessity for synchronization and communication between these modules: more protocols.

Today's information-handling systems or networks are frequently put together from subsystems from many different sources. The component systems — be they from a single supplier or from different suppliers — must function well together. It is hardly necessary to say that the suppliers must have identical interpretations of the protocols they are to implement. Prose definitions have been found to be far too imprecise; thus, there is a growing reliance on formal specifications.

Protocols should be free from errors. The use of formal specifications opens the door to *validation* — showing that a protocol is free from certain kinds of syntactic errors — and to *verification* — showing that a protocol provides some specific function.

Once implemented, a protocol should perform reasonably well. It is important to have an indication of *performance* early in the protocol design process.

The implemented protocol must be a faithful instantiation of the specified protocol. Conceivably this could be guaranteed by a proven *compilation* procedure, driven direct

from the formal protocol specification and delivering code in the target-machine language. Failing such an (unlikely) procedure, the protocol implementation should be *tested* or *certified* relative to the formal specification.

Since there are many protocol standards both planned and already in use, it is necessary to provide *conversion* from one system of protocols to another. The first theoretical work has just been published in this area.

These requirements lead to the topics covered in this paper: 1) formal protocol specification; 2) trying to ensure that these specifications are error-free; 3) showing that the protocol architected will perform reasonably well when implemented; 4) producing an implementation of the protocol which corresponds to the formal specification from which it is derived; 5) testing the implementation for conformance to the architecture defined in the formal specification; and 6) designing a machine for conversion of one protocol to another.

These same topics form the basis of a system approach to "protocol engineering", a term coined by Tom Piatkowski in 1981 (Piatkowski81). This author's view of such a system is shown in Fig. 1. We shall now discuss, in sequence, the aspects of protocol specification and analysis just enumerated.

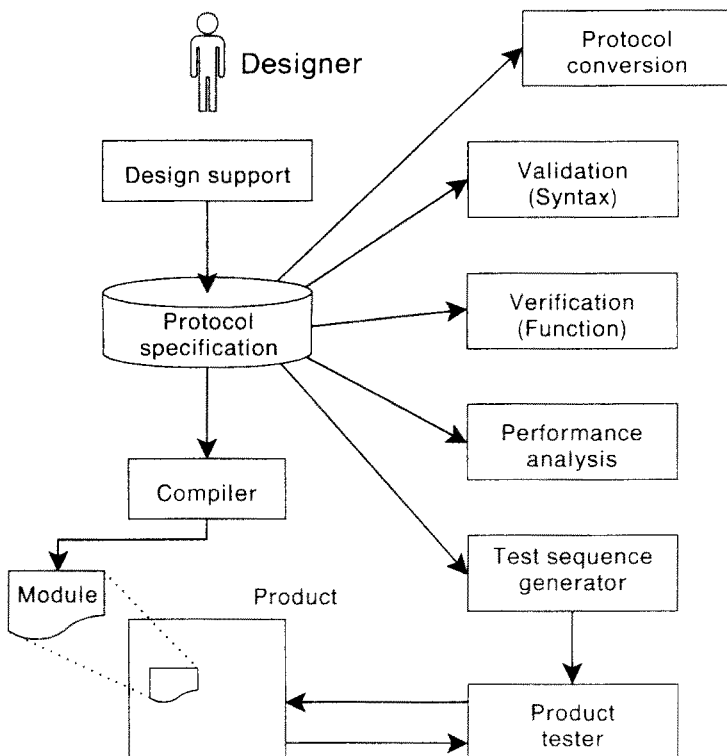


Figure 1. One view of protocol engineering

## II. FORMAL DESCRIPTION TECHNIQUES

There is a number of formal description techniques in use for formally specifying protocols and still more have been proposed. The author's experience has been mainly with finite-state-machine (FSM) based descriptions which have also evolved as the most practical and most widely used. After a discussion of FSM-based description techniques — the clear emphasis in this paper — some alternative approaches will be sketched.

### II.1 A simple finite-state-machine model

It is easiest to discuss formal protocol specification on the basis of a concrete example. Perhaps the simplest, formally specified standard protocol is the logical (third) level of the X.25 specification (CCITT84d). The formal part of the X.25 specification is based on a finite-state-machine description and is shown in Fig. 2.

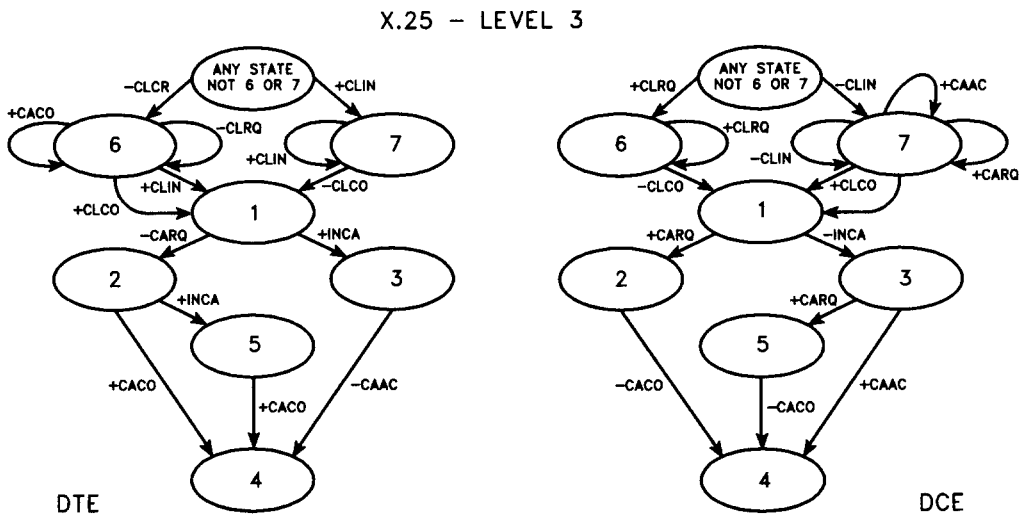


Figure 2. The X.25 specification. From (Rudin86b), Copyright © 1986 by IEEE Switzerland Chapter on Digital Communication Systems

Fig. 2 shows two processes which can communicate with each other. Ovals represent the state of the user equipment (DTE) on the left and of the network-access equipment (DCE) on the right. The arcs connecting states are labeled with an event or message name preceded by a (-) sign to represent transmission of a message to a partner or by a (+) sign to represent reception of a message from a partner. The initial state for both processes is No. 1. If the user (left diagram) wanted to initiate a logical call he would transmit the message -CARQ (Call Request), so transiting from State No. 1 to No. 2. The network would receive this, +CARQ arc, and also transit from State No. 1 to No. 2. If the

call could be set up to the destination, the network would ultimately transmit CACO (Call Connected) to the initiating user. After the DTE has received this CACO message, both partners are in State No. 4 and ready for transmission of useful data.

It is also possible for the network (DCE) to signal the user that he has an incoming call (—INCA), to which message he should respond by signaling his acceptance of the incoming call (—CAAC). At the top of Fig. 2 are paths and states used to terminate the connection and, again, with the termination initiated either from the DTE or from the DCE.

The transmission of messages — again, the arcs labeled (—) — are asynchronous and can essentially take place at any time the process being described is in the state just preceding such an arc. In general, but also in the case of the X.25 specification, it is this asynchronicity which leads to many possible event sequences. Usually there are so many that the designer cannot maintain a complete view of the possibilities, and errors are introduced. More of this later.

The description of the X.25, Level-3 protocol above is in the form of a simple finite-state-machine (FSM) description. X.21 is similarly described. When protocols become more complex, the style of Fig. 2 becomes unwieldy. For such protocols the most popular description is the extended-FSM one.

## II.2 Extended finite-state-machine models

One reason for the incorporation of extensions is that most protocols are so complex that it is impractical and certainly unwieldy to record all the necessary information to describe the dynamics of the protocol as different explicit states. Rather, the information conceptually less important — such as addresses and sequence numbers — is stored as auxiliary variables. Decisions can be based and actions can be performed on these auxiliary variables. This results in an "extended" FSM description.

Fig. 3 is a fragment of an extended-FSM description. There is only one transition arc leaving the upper state so that no predicate (P: ) is needed. The action (A: ) associated with this arc is first to transmit the message HEADER and then to set SQNO to 0. The next arc to be followed is determined by the two predicates, i.e., the value of the variable SQNO. If  $SQNO < N$ , then the arc is followed which loops back to the lower state, SQNO is advanced by one, and the message INTERMED is generated. Finally, when  $SQNO = N$ , the other departing arc is followed and the message TAIL is generated.

CCITT has had an effort for some time to create SDL (Specification and Description Language) which is an extended-FSM language (Dickson83). Experience had shown that while the formal part of the X.21 and X.25 definitions was very useful, these formal definitions could only be interpreted with heavy reference to the text which accompanied them. SDL improves this situation inasmuch as more of the protocol description can be

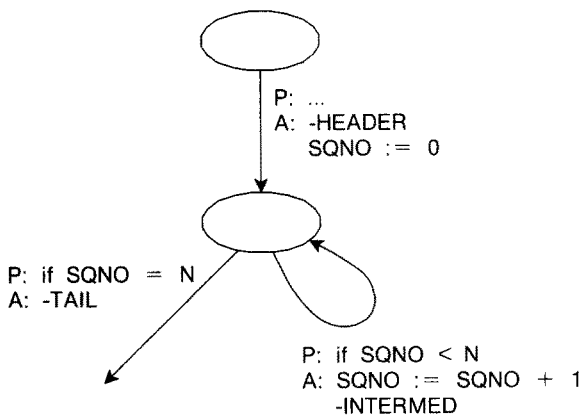


Figure 3. A sample extended-FSM description

formalized. A refinement of SDL in a Pascal-oriented language is being proposed within CCITT as Recommendation X.250.

ISO's work on formal description techniques (FDT's) in connection with the Open System Interconnection (OSI) Architecture (Zimmermann80) is proceeding in three parts. Subgroup A is concerned with architectural concepts, to be used in turn by the other two subgroups, concerned with FDT languages, per se. The concepts of "modules" which contain the description of the function of the protocol, and the "channels" which provide communication between the modules have been specified.

An important concept in any structured design is the notion of consecutive refinement. With this concept a high-order module can be defined which simply specifies that some function be carried out. As the specification develops, such a module may be refined repeatedly, decomposing the function at one level into the subcomponent functions required at a lower level, thus increasing detail at every refinement step. This process facilitates a high-order overview of the entire system at one end and all the details required for compatible implementation at the other. This concept is also included in the work of Subgroup A (Vissers83).

Subgroup B is working on "ESTL", Extended State Transition Language, which is an extended-FSM approach. ESTL, also called "Estelle", is based on the language Pascal. Various features have been added, particularly those which facilitate definition of finite-state machines (Vissers83).

The language IBM uses for describing SNA is called FAPL (Format and Protocol Language). It is derived from PL/I and like Estelle contains additional constructs for handling finite-state machines and processes (Pozefsky82).

An important question in any formal description technique is "To what degree should the specification affect the implementation?" Clearly implementations of the same architecture

must be compatible but how many hints should the architects place in the specification to influence the implementers direct? SNA provides many such hints via FAPL and even calls its FAPL definition a "Meta-Implementation". The definition itself comes close to a real implementation. This has the advantage that the definition can be executed and many questions about how the architecture performs can be answered by exercising the definition itself. Parts of the implementation are deliberately left to the implementers and appear in the formal specification only as "undefined protocol machines".

### II.3 LOTOS

Subgroup C of ISO's effort on formal description techniques is working on an approach which has its roots in a formally complete theory of the sequence of messages exchanged by communicating processes: R. Milner's Calculus of Communicating Systems (Milner80). The language being developed by Subgroup C is LOTOS (Language for Temporal Ordering Specification) (Vissers83) and (Brinksmas85). Since an effort is made only to describe the message sequences, there is a minimum impact on the specification of an implementation. This leaves implementers the maximum amount of freedom but still provides sufficient guidance to ensure compatibility.

It is instructive to express the protocol fragment shown in Fig. 3 in LOTOS:

```

process P[a] (N) :=
    (* Offers at gate "a" one copy of message HEAD, *)
    (* N copies of INTERMED, one copy of TAIL *)
    a !HEAD ; Q[a] (N)
where
    process Q[a] (m) :=
        [m=0] -> a !TAIL ; stop
        []
        [m>0] -> a !INTERMED ; Q[a] (m-1)
    endproc
endproc

```

The definition is composed of a process **P** to which the parameter **N** is passed, and a process **Q**, called by **P**. Process **P** first offers the message **HEAD** at gate **a** and then calls process **Q**. Process **Q** takes one of two actions depending on the value of the parameter **m**, with which **Q** is called. If **N** messages **INTERMED** have already been offered, the value of **m** is zero, the message **TAIL** is offered and the process halts. Otherwise the message **INTERMED** is offered and **Q** calls itself recursively, passing the parameter **m** reduced by one.

The greatest promise of LOTOS lies in the fact that it allows as many levels of refinement as are needed, through the use of two language operators "parallel composition" and "restriction". This is not the case for Estelle. On the other hand, LOTOS appears to be

more abstract, compared to FSM-based languages. The names of states in an FSM description provide a convenient “crutch” for human interpretation.

#### II.4 Other formal approaches

There are a number of other approaches to formal specification. Of the ones not yet mentioned here, the Petri net is probably the most popular (Diaz82) and (Diaz83). Petri nets have rarely been used for the specification of protocols; their main utility is in the analysis of protocols rather than a means of disseminating protocol specification. Petri nets consist of *places* and *transitions* to represent conditions and events. A very readable general introduction is given in (Peterson77).

Fig. 4 shows a fragment of a sample Petri net modeling part of a transmitter on the left and part of a receiver on the right, connected by a communication medium between the dashed lines in the center of the figure. The initial *marking* — corresponding to the initial state of the system — has *tokens* in the two READY places. The *enabled* transition (it has a token in all of its input places) at the upper left may *fire*, removing the token from the upper-left READY place and putting tokens into the other place at the left and in the place labeled Message in the channel. At this point the upper transition in the receiver is enabled and may fire putting a token into the lower place in the receiver. Firing of the lower transition in the receiver puts a token into the place in the channel labeled Ack. Firing of the lower transition in the transmitter, in turn, models successful receipt of the acknowledgement by the transmitter.

Hybrid approaches — combining different formal description techniques — have also been suggested. A recent example is (Sabnani85) wherein a combination of FSM- and temporal-logic-based formal description techniques is used.

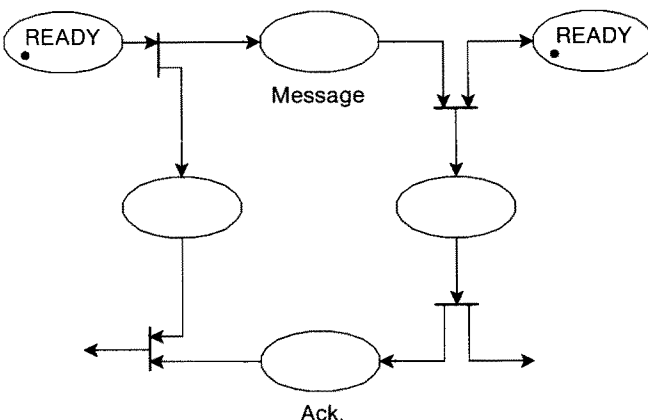


Figure 4. A fragment of a simple Petri net

It will be some time before the pros and cons of all these different formal description techniques are known. The only way of evaluating these is to use them on various protocols and this is just what is happening at the present time. There is a great deal of cooperation, particularly among CCITT and ISO efforts. Comparison of a number of formal description techniques can be found in (Venkatraman85).

It is quite clear that a formal protocol specification is mandatory since only in this way can one be sure that all the implementers have the same mental picture of the protocol they are implementing. In the following we assume that such a formal description is available, and further that it is machine readable — as are all the techniques listed above. So given such a formal specification for purposes of communicating the protocol specification to its implementers in an unambiguous fashion, what else can be achieved?

### **III. FUNCTIONS BASED ON A MACHINE-READABLE SPECIFICATION**

In the following it is assumed that the formal protocol specification is also machine-readable. Given that assumption some of the functions shown in Fig. 1 can be obtained in semi-automatic fashion. Ideally, a designer should express his protocol formally, be able to test this specification for correctness (validation and verification), obtain some early indication of how it would perform, compile major parts of the implementation direct from the formal specification, and finally test the resultant implementation to assure that it conforms to the specification. In order to provide interworking between two different protocol systems, conversion is required. Let us now look at these various functions and the state-of-the-art.

#### **III.1 Eliminating Errors from the Specification**

We first look at some of the errors which occur in protocols: those of syntax (or form) and those of semantics (or function).

##### **III.1.1 Validation**

A test for syntax, usually called validation, can be performed on protocols. Some of the kinds of errors which can often be automatically detected are unspecified reception, unexercised code, and static deadlock, when the specification is of the finite-state-machine type (Zafiropulo80). What are some of these errors?

Reverting to Fig. 2 suppose that the DTE were in State No. '1 and sent a call request CARQ to the DCE and at the same time the DTE were in ANY STATE NOT 6 OR 7 and sent a clear indication CLIN to the DTE. "At the same time" means, for example, that the DTE sent the CARQ after the DCE had sent CLIN but before the DTE had received it. (Part of the

abstraction of this model is that it can take some time for messages to travel between the communicating processes; the processes are in effect connected by queues. The actual time delay introduced by these queues, per se, and the permutation of message interleavings made possible by a random distribution of delays are what make protocol analysis difficult.)

Thus there has been a collision of messages. An inspection of Fig. 2 reveals that the DTE can receive the CLIN in State No. 2 because this is also ANY STATE NOT 6 OR 7; the only way the DCE can receive the CARQ message in State No. 7 is via the arc which loops back to State No. 7 labeled +CARQ. Exactly this latter arc was missing from the 1976 specification (CCITT76) and constituted an *unspecified reception* error. The error has long since been corrected (CCITT81). This unspecified reception type of error is the most common error we have seen and in experience occurs in most protocol designs. The 1976 version of the X.21 Specification also had errors of this type, again, long since corrected (West78a).

It is also possible that during the evolution of a protocol design, part of the specification becomes inaccessible. The resulting *unexercised code* can also be automatically identified.

It has also occurred that, given a particular sequence of message interchanges, all further interaction among the protocol partners becomes impossible. Each protocol process is waiting for a message from one of the other processes and there are no messages underway in the communication medium. This constitutes a *static deadlock* and can also be automatically flagged via validation.

When the protocol specification is in an executable form, validation can be carried out by exercising the protocol partner models with each other as shown in Fig. 5. The driver systematically forces the protocol partners into every conceivable situation, where "situation" includes not only the states of the communicating partners but also the state of a

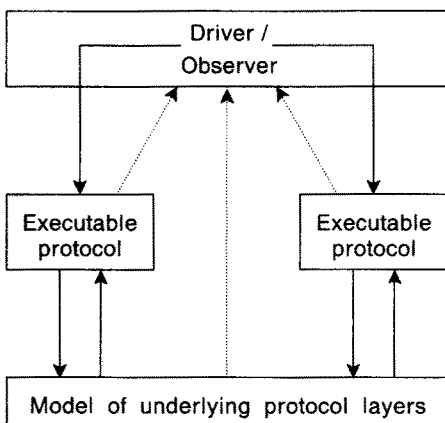


Figure 5. Protocol validation

model of the communication medium — and underlying communication protocols, if any. The different situations can be recorded as a global state and, if the protocol is not too complex, all of these global states can be found and examined to detect the error types mentioned above. West (West78b) has described a software package which efficiently performs this validation. Techniques will be further described in Section IV.

The same approach may be used to determine a lack of synchronization among the protocol processes involved by detecting potential *state ambiguities* (Schultz80).

It is also possible for a protocol to enter into a nonproductive operation cycle, in which nothing useful occurs and from which the protocol cannot extricate itself. Examples are the various packet-retransmission deadlocks found in early packet-switching networks. A packet arriving at a node consistently finds the storage buffer of that node full so that the packet has to be discarded. The node from which it came, after a recurring timeout period, hopelessly retransmits the packet. These “livelocks” or *dynamic deadlocks* can also be automatically detected (Sherman82).

*Time*, for example the value of the timeout specification in the example just given, plays an important role in protocols. Validation of a protocol should include a validation of the complete protocol specification, including time values for timeouts. The inclusion of time brings increased complexity to an already difficult problem. An approach for handling the validation of a protocol including the specification of time is (Bolognesi84).

One of the advantages of validation is that it is applicable to virtually all protocols. Because we are dealing with parallel processes which may interact in many ways, it is difficult for human designers to anticipate all the different interaction sequences and produce an error-free design. The author and his colleagues have used or seen validation tools used on twenty-odd protocols and only one of these was error free (Rudin82c)! All the rest had errors of varying degrees of severity.

### III.1.2 Verification

Validation helps to ensure lack of syntactical errors, errors which would prevent the protocol from fulfilling its function. But, even in the absence of syntactic errors, will the protocol achieve its functional objective? This is a question of protocol verification.

An example from the author’s experience occurred in the analysis of a token-ring local-area network. Here a special station on the ring — the monitor — ensures that transmission noise would not cause mutilation, corruption, or loss of a token in such a way that the system would be unable to recover. An automated verification of the system, using state exploration, was performed (Rudin82a) and (Rudin82b) showing that the system architected did indeed have this desirable property.

The verification process just described was tuned to the token-ring protocol at hand. In contrast, there is a number of semi-automatic, generally applicable verification packages

available. These and experience with these program packages are summarized in (Sunshine83). Methods of calculus, in the form of partially automated, interactive theorem provers, show promise for the verification of protocols which would be difficult to handle using state exploration techniques. An example of this is work in progress at the University of Karlsruhe (Krumm84). Temporal logic shows promise for complex protocols; a recent survey is (Hailpern85). A comparison of using three different techniques for detecting errors in a protocol is given in (Groz84). The methods applied were simulation (effectively, prototyping), state exploration (of a Petri-net model), and logical proof.

A problem which recurs in trying to identify errors in complex protocols is the sheer volume of the information required to describe the dynamics of the protocol. When using exhaustive state-enumeration techniques this manifests itself as an explosion of the state space to practically unmanageable size. Lam and Shankar have proposed a solution to this problem using the method of projections (Lam82). The idea here is to project the information required to describe the dynamics of the protocol into a space where only the minimum amount of information is required. This property has been successfully applied to several protocols, including time-dependent protocols (Shankar82).

We end this section with a list of the error types mentioned, detection of which can be partly automated when the specification is of the executable type.

**Table I - Error Types**

**Syntactic errors (found via Validation)**

- Unspecified reception
- Unexercised code
- Static deadlock
- Dynamic deadlock
- State ambiguity

**Semantic errors (found via Verification)**

- Functional properties

At this point in our design system we have a formal, machine-readable definition of the protocol. The process of generating the formal definition has itself more than likely prevented numerous errors. By using validation we have removed some syntactic errors and by using verification have demonstrated that the protocol does in fact provide at least those specific functions for which it has been examined.

### III.2 Performance Prediction

How well will the protocol perform in terms of throughput or response time? Until recently there were two main means of ascertaining performance. One was to make a model of the protocol and its environment and then to use various mathematical tools — such as traffic theory — to make a performance analysis. The other possibility was to simulate the protocol and then to gather statistics on the performance of the model simulated. The availability of a machine-readable description of the protocol ought to be, and is, of some help.

First it is necessary to augment the formal description in order to specify the elapsed time anticipated and the frequency with which alternate paths may be taken. Fig. 6 shows an FSM model of a nonideal communications medium. The style of the model follows (Bolognesi84). From a transmitter (T) — not shown — the channel receives a message, the +MESS arc, and in so doing transits from State No. 0 to State No. 1. Following this, the channel either loses the message (−NULL), faithfully retransmits it (−MESS), or retransmits it in error (−ERROR) to the receiver process (R) — not shown either. The time-duration labels indicate that the process may spend an arbitrarily long time in State No. 0, specifically until it receives MESS. Once in State No. 1, however, the process must spend at least a time of duration  $t_{min}$  and must exit (by taking one of the specified actions) before a maximum time duration of  $t_{max}$ .

The first work to predict performance making use of a formal specification is that of Bauerfeld (Bauerfeld83) wherein he generates a simulation model from the formal description automatically. A more direct approach was explored by the author (Rudin84) wherein performance estimates are obtained direct from analysis of the global finite-state diagram of the protocol. This latter analysis is based on an FSM description. Similar

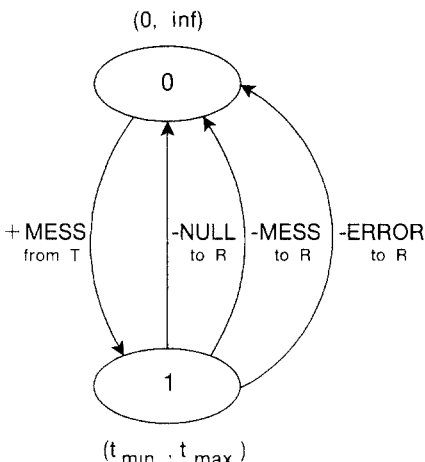


Figure 6. FSM model of nonideal channel with time and branching possibilities

results have been obtained based on Petri-net descriptions (Molloy82) and (Razouk84). Very recently an effort has been made to combine the techniques of queueing network theory in an analysis driven direct by the formal specification (Kritzinger84b). This latter work is directed specifically at the OSI Reference Model and takes into account the interplay among various protocol layers as well as the coexistence of several protocol instances at any one layer.

An example of the kind of results which may be had, for the case of a data link control protocol, is shown in Fig. 7 after (Rudin84). This figure shows the average delay between successful receptions of messages (including retransmissions resulting from message loss or message corruption in a communications medium similar to that shown in Fig. 6).

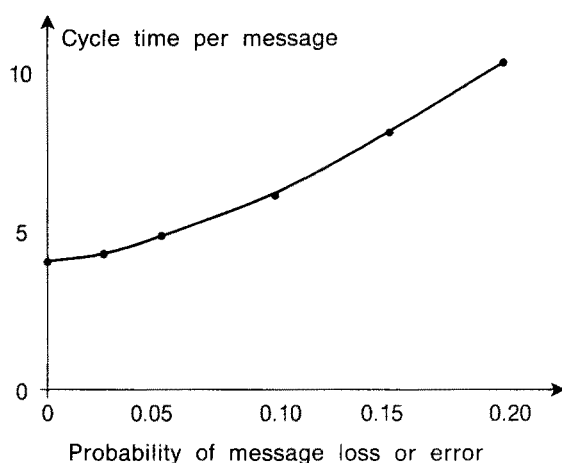


Figure 7. Delay in a simple data link control protocol

### III.3 Automated Implementation

The final objective of any design is incorporation of the protocol in a successful implementation. Traditionally, this has been done by hand-coding. Again, the availability of a machine-readable specification opens the door to direct compilation of parts of the code required for an implementation. In the last few years, direct compilation of a protocol has been carried out in several instances. There will always be substantial portions of an implementation which must be hand-coded but there is the hope of being able to produce the bulk — say 60 percent — of the necessary code by automated means.

In the case of SNA, Nash has demonstrated automated generation of an implementation in work he did in 1979 (Nash83). Compilation of a protocol — in this case part of SNA's Data Flow Control — was carried out in two major steps. First the FAPL compiler was used to expand the FAPL specification into an intermediate language (PL/S). The manual code required was also written in PL/S. This PL/S code was then compiled and assembled into

the machine code required. The two-step process means that only one FAPL compiler is required and takes advantage of existing PL/S compilers which produce code for various target machines. The particular application was for part of the IBM 8100 Information System; the results in terms of path length required (number of instructions to be executed) and memory space required for the code were satisfactory (Nash83) and (Smith83). The use of an automated technique substantially reduced implementation time. Automatic compilation has been used several times in the case of SNA (Nash83) and (Smith83).

The NBS (National Bureau of Standards) has developed its own extended-FSM language, actually a predecessor and subset of Estelle. The NBS language was used to describe a subset of the OSI File Transfer Protocol as part of a test tool to be described in the next section (Linn84) and (Mills84). The point here is that for the test tool developed at NBS, a reference model, i.e., an executable model of the file-transport protocol was required. The formal specification was compiled into the language "C". About 40 percent of the code required in "C" could be automatically compiled; the remainder had to be hand-coded.

#### **III.4 Testing for Architectural Conformance**

It is unlikely that the complete protocol specification can ever be compiled direct into a software or firmware implementation; there are simply too many idiosyncrasies for each particular microprocessor engine. The result is that there will always be a need for techniques to test an implementation to demonstrate that it does in fact correspond to the formal description from which it was derived.

An early use of a formal protocol definition for testing occurred in conjunction with SNA in the form of a tool called METROPOL (METHod of Testing Real-Operation Protocols Off-Line) (Cork83). In METROPOL dialogs between a prototype implementation and its partner are derived from trace recordings made during conventional test sessions. The message sequence derived can then be used as input to the executable FAPL definition of SNA. SNA is rich in its internal checking function and any error detected via these checks — as well as any other anomalies observed — can be flagged to the operator. METROPOL found a number of errors in prototype SNA implementations.

The National Physical Laboratory in Teddington has long been active in developing protocol assessment techniques (Rayner83). As a result of the work done at NPL, a pilot assessment center has been set up at the National Computing Center in Manchester. In this approach the implementation under test is exercised by a test responder and an "encoder-decoder", capable of applying normal (protocol-specification conform) sequences as well as abnormal sequences to the implementation under test. The abnormal sequences are used to test the response of the implementation to error conditions.

In a similar approach the National Bureau of Standards (NBS) in the US uses the formal specification more direct as a reference implementation with which the implementation

under test communicates; an exception generator is used to generate abnormal sequences (Linn84) and (Mills84). NBS has also used the formal protocol definition to generate automatically some of the sequences or "scripts" used to drive both the reference and test implementations in a manner similar to that indicated in Fig. 1.

The test arrangement is essentially that of Fig. 8. In validation, one executable model of the protocol is tested against another in a search for what might be called logical inconsistencies. In testing for architectural conformance the (implemented) protocol under test is tested against the executable definition of same. The protocol under test is driven by a remote driver which represents the user and/or higher-level protocols. The reference protocol also has a driver. Geographical separation of the reference protocol and the protocol under test poses difficult coordination problems, particularly when an attempt is made to determine errors which are sequence-interleaving dependent. An exception generator can be used to simulate errors in the communicating medium, which itself — along with the lower-level protocols — appears at the bottom of the figure.

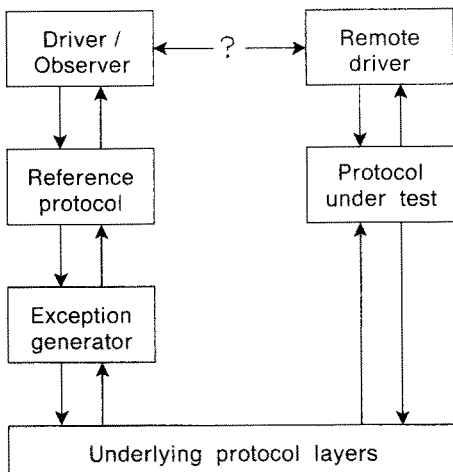


Figure 8. Testing for architectural conformance

Automated testing systems — and the importance of formal protocol definitions as well — were recently given credence in a demonstration at the National Computer Conference held in Las Vegas in July 1984. The demonstration was coordinated by NBS and based on the ISO Class-4 Transport Protocol. About a dozen manufacturers successfully demonstrated communication among their equipment using an agreed file-transfer protocol as the common application. The formal definition and the automated testing of the various implementations played a key role in the success of the demonstration (Linn84).

The testing of protocols — aided in part by the current desire to be able to interconnect heterogeneous systems — has recently become an area of intensive effort. Standards for testing OSI protocols are being developed (Rayner85).

### III.5 Protocol Conversion

Because there are so many different user needs and so many different network environments, there are many different protocol standards. Some of these have been in use for years while others are being developed and others are yet to be conceived. This multiplicity of protocols causes difficulties when the need arises to send information from one network to another and the two networks embody different protocols.

For this case one needs to provide conversion between the two protocol systems. The problem and ad hoc solutions to it are described in (Green86). The area of protocol conversion is the most recently established area in protocol engineering.

Fig. 9 illustrates the difficulty. At the top left, is a computer-communication system based on protocol system A. The one at the top right, is based on protocol system B. At the bottom, conversion — symbolized by a question mark — is required so that a terminal using protocol B can communicate with a host using protocol A.

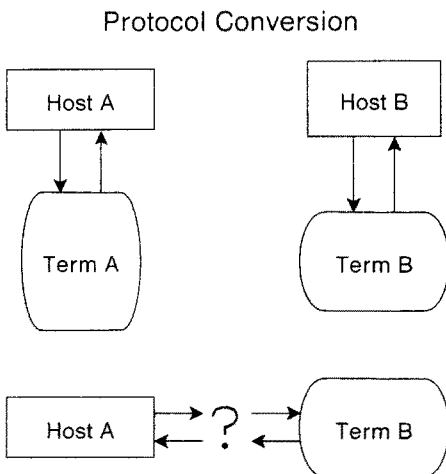


Figure 9. The protocol conversion plane

There are two papers which lay the beginnings of the theoretical basis required to semi-automatically design a mechanism capable of providing the conversion function. Lam uses the method of projections in his approach to the problem (Lam86). Okumura is able to first determine whether a converter exists and then to provide a construction method for a

converter, provided that both protocols are specified as simple finite-state machines (Okumura86). Much effort is required before such techniques approach practicality.

### III.6 Change Management

Change management is not a concept which appears explicitly in the protocol engineering system of Fig. 1. Consideration of the problem of managing change, as additions and improvements to the protocol system architecture are made and new products are introduced, results in an additional factor in favor of formal, machine-readable specification. To the extent that the procedures described above have been automated, the need for manual work to re-do the various functions is minimized.

## IV. SOME NOTES ON TECHNIQUES

In the work on validation, verification, and performance prediction with which the author has been associated, state-exploration techniques have played a key role. The validation technique we have used most, generates a tree of global states (West78b). A global state consists of the state of each of the component processes plus the ordered content of the communication media or communication channels (modeled as queues of messages or events), which may pairwise interconnect the processes.

This construction of a global-state tree, rooted in the global state where each of the component processes is in its initial state and all the communication channels are empty, is shown schematically in Fig. 10.

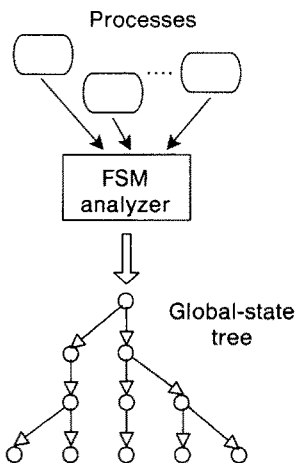


Figure 10. From the FSM representation to a global-state tree

Starting from the initial state, all admissible successor states are identified. New global states are added to the tree; growth stops when no unknown global states can be generated. For each of the leaves in the tree, a pointer is kept to the global states (already generated) which could follow. This global-state tree is the basis for most of the analyses we have made.

The process of validation can be achieved incrementally during the generation of the global-state tree or can be thought of as being driven from the completed global-state tree. The latter is shown schematically in Fig. 11. Each one of the global states is checked to see whether it is free of unspecified reception errors (where one of the processes is offered a message from a channel in a state in which no provision has been made for its reception), does not result in a deadlock, or is unreachable (indicating unexercisable code). Again the reader is referred to (West78b).

Fig. 12 shows one means of verifying that a protocol provides some specified function. Again the process relies on state exploration and is driven from the global-state tree. The technique used in (Rudin82a) and (Rudin 82b) to demonstrate that a token-ring local-area network would quickly recover from several different kinds of transmission errors used the method of Fig. 12. First, all the global states in the global-state tree were found which had the property that an error had just been introduced. (Errors were introduced by an artificially added "demon" process.) Paths were then generated starting from each one of these global states and continuing until a global state had been reached which indicated that error recovery had taken place successfully. Count is kept of the number of iterations required to generate the paths; when this count exceeds a preset limit, the conclusion is that recovery has not yet occurred or may never occur.

Finally, Fig. 13 indicates one means of predicting protocol performance, again driven from the global-state tree and using state-exploration techniques. To be able to determine

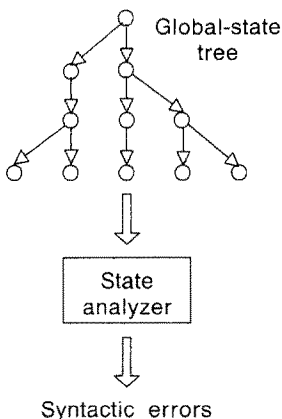


Figure 11. Validation: Detecting syntactic errors

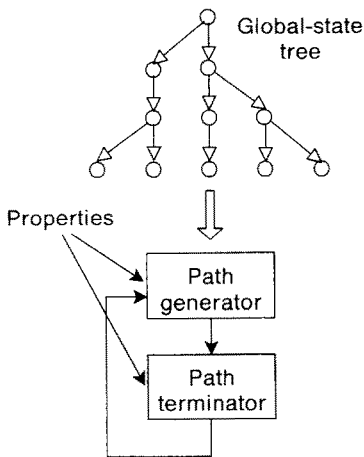


Figure 12. Verification: Proving function

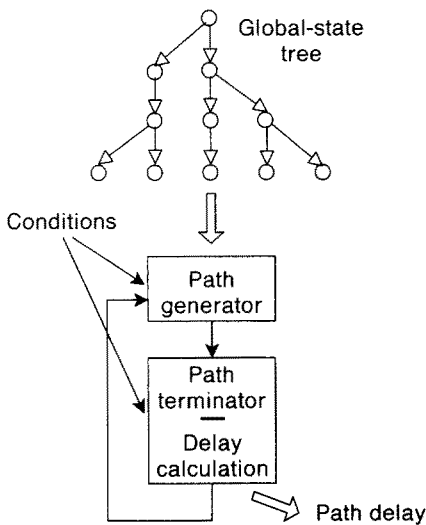


Figure 13. Predicting protocol performance

performance the finite-state-machine description must be augmented, as discussed in connection with Fig. 6. Delays (usually average delays representing the time it takes to execute a transition) and branching probabilities (when one of several transitions may follow a global state) are added to the formal specification.

To calculate average delay, paths can be explored which emanate from a global state with particular properties (the reception of a positive acknowledgement in the data-link-control protocol for which performance is shown in Fig. 13) and terminate in some

global state (the identical state in the case of the protocol of Fig. 13). The average delay may then be calculated as

$$\text{Average delay} = \sum_{i=1}^N (P_i)(D_i),$$

where  $D_i$  is the delay along a path and  $P_i$  is the probability of being on that path.

The calculation is terminated when the sum of the probabilities of unterminated paths has dropped below a specified threshold.

It has been recently shown that the same technique can be used to determine the complete distribution function of delay with little increase in complexity (Rudin86d).

## V. FUTURE DEVELOPMENTS

In our tour of the protocol engineering system of Fig. 1, the reader may have been misled into thinking that all the problems have been solved. This is not the case; there is need for continuing research in all the areas mentioned. It should also be stated that several of the examples given are taken from work which was close at hand to the author; there are alternative approaches, some of them with advantages over the methods presented here, depending on the circumstances. Pointers to the literature covering some of these alternatives are given in Section VI.

Let us now review the areas discussed to see some of the further research needed.

### V.1 Validation

We are unable to validate most "complex" protocols. The most prevalent technique is exhaustive exploration of a multidimensional state space. A "complex" protocol leads to an impractically large number of states when validation is attempted. Many protocols are complex and the need for an appropriate approach is urgent.

A promising and recent development to circumvent this problem is the notion of a colleague, Colin West, to use a driver which randomly, instead of deterministically, explores state transitions (West86). The technique has been applied with success to the OSI Session Level Protocol (West86). It has also been applied to testing protocol implementations. In this application J. Rubin let his implementations run long enough to generate over a million messages on each side and found several dozen errors of distressing subtlety (Rubin85).

A related question deals with partitioning: "Under what conditions is it satisfactory to validate the component parts of a protocol, having a guarantee that a validation of the complete protocol is unnecessary?" Further exploration of the calculus underlying LOTOS

will likely shed some light here. The same is true for Lam and Shankar's method of projections (Lam82).

Often it is necessary to include specification of one or more time-outs in a protocol. A difficult problem is validation of a protocol to determine whether or not these values are consistent with each other and with the overall protocol specification. As mentioned above, work in this area has started (Bolognesi84) but much more is needed.

## **V.2 Verification**

When exhaustive exploration is used as a basis for verification, the problem of handling complex protocols again arises. Alternatively there are algebraic techniques but these tend to require heavy human intervention. There is a need both to refine these techniques and to automate them to a greater extent. The use of hybrid specification and analysis techniques as, for example, in (Sabnani85) is promising.

## **V.3 Performance Prediction**

Work here, in the sense that performance analysis programs are driven direct from the formal specification of the protocol, has just begun. One interface which promises to be fruitful is that between traditional queueing theory with its ability to take waiting customers into account and the contention-free analysis discussed above.

## **V.4 Automatic Implementation**

While there have been a few experimental efforts here, experience is relatively limited, despite the obvious large return. Using the latest techniques in compiler design, a number of test cases could be made comparing the memory and execution-time efficiencies of automatically and manually produced implementations.

## **V.5 Automated Testing**

How long should a test be run? How much test coverage is obtained at the cost of what execution time? Is it possible to design particularly effective test sequences? What is the point of diminishing returns in testing? There are many open issues here. Experience with the first automated test systems is just becoming available. This experience will help to determine future directions.

## **V.6 Protocol Conversion**

As stated earlier, this topic is in its infancy. The first two publications dealing with automated techniques for doing conversion appeared only this year (1986). As the desire to communicate using computer-based messaging systems increases, the demand for conversion will grow.

After this catalog of unsolved problems, the engineering design system shown in Fig. 1 may seem rather remote. Suffice it to say that one organization is beginning to sell components of just such a system (PDC85) based on protocols specified in Estelle language.

## VI. POINTERS FOR FURTHER READING

The December 1983 issue of the Proceedings of the IEEE (Folts83) is devoted to the OSI Model and related topics. A special issue of the IEEE Transactions on Communication in April 1980, since published with additions in book form, is a collection of papers on computer-networking architecture and protocols (Green80). Carl Sunshine also edited a collection of papers, including some of the early classics in protocol definition and analysis (Sunshine81).

A number of the topics emphasized in this paper was covered in a session at the ICC83 in Boston (Bauerfeld83, Diaz83, Nash83, Piatkowski83, Rayner83, Sunshine83, and Tenney83). Finally, IFIP Working Group 6.1 has sponsored a series of annual Workshops devoted to the specification, verification, and testing of protocols which contain state-of-the-art contributions (Rayner81, Sunshine82, Rudin83a, Yemini84, Diaz85, Bochmann86). The next of these Workshops is scheduled for Zurich in May, 1987.

## VII. CONCLUSION

Formal protocol specification is a concept which has firmly taken root and is here to stay. There is likely no "best" formal specification technique. Further work is required to understand where each of the existing techniques has its greatest advantage. There is scope for much more work on the tools which may be driven from a machine-readable formal specification. The success of the tools which already exist promises handsome rewards for this future work.

## VIII. ACKNOWLEDGMENT

I thank my colleagues at the IBM Zurich Research Laboratory for innumerable discussions on protocols. Particular thanks go to Tommaso Bolognesi for many suggestions and for the protocol fragment written in LOTOS.

## REFERENCES

Bauerfeld83- W. L. Bauerfeld, "Protocol performance prediction," Proc. International Conference on Communications, IEEE, Boston, Mass., June 20-23, 1983, pp. 1311 - 1315.

Bochmann86- G. v. Bochmann and B. Sarikaya, Eds., Proc. Sixth IFIP Workshop on Protocol Specification, Testing, and Verification, VI, Gray Rocks - Montreal, June 10-13 1986, (to be published by North-Holland, Amsterdam).

Bolognesi84- T. Bolognesi and H. Rudin, "On the analysis of time-constrained protocols by network flow algorithms," Proc. Fourth Workshop on Protocol Specification, Testing, and Verification, Skytop, Pennsylvania, June 1984, (North-Holland, Amsterdam, 1985), pp. 491 - 513.

Brinksma85- E. Brinksma, "A tutorial on LOTOS," Proc. of the Fifth Workshop on Protocol Specification, Testing, and Verification, M. Diaz, Ed., Moissac-Toulouse, June 10-13 1985, (North-Holland, Amsterdam, 1985), pp. 171 - 194.

CCITT76- C.C.I.T.T., Orange Books, Sixth Plenary Assembly, Vol. VIII.2, Geneva, 1976.

CCITT81- C.C.I.T.T., Yellow Books, Seventh Plenary Assembly, Vol. VIII.2, Geneva, 1981.

CCITT84d- C.C.I.T.T., Red Book, Volume VIII - Fascicle VIII.3, Data Communication Networks Interfaces Recommendations X.20 - X.32, VIIIth Plenary Assembly, Torremolinos, October 8-19, 1984.

CCITT84s- C.C.I.T.T., Red Book, Volume VI - Fascicle VI.10, Functional Specification and Description Language (SDL), Recommendation Z.101- Z.104, VIIIth Plenary Assembly, Torremolinos, October 8-19, 1984.

Cork83- R. M. S. Cork, "The testing of protocols in SNA products," Proc. of the Workshop on Protocol Specification, Testing, and Verification, III, Rüschlikon, Switzerland, May 1983, (North-Holland, Amsterdam, 1983) pp. 455 - 463.

Diaz82- M. Diaz, "Modeling and analysis of communication and cooperation protocols using Petri-net based models," Computer Networks, Vol. 6, 1982, pp. 419 - 441.

Diaz83- M. Diaz, "Status of using Petri nets for protocols," Proc. of the International Conference on Communications, Boston, Mass., June 20-23, 1983, pp. 1301 - 1305.

Diaz85- M. Diaz, Ed., Proc. of the Fifth International Workshop on Protocol Specification, Testing, and Verification, Moissac-Toulouse, June, 1985, (North Holland, Amsterdam, 1985).

Dickson83- G. J. Dickson and P. J. Chazal, "Status of CCITT description techniques and application to protocol specification," Special issue of the Proc. of the IEEE on OSI, Vol. 71, No. 12, December, 1983, pp. 1346 - 1355.

Folts83- H. C. Folts and R. desJardins, Eds., Special issue of the Proc. of the IEEE on OSI, Vol. 71, No. 12, December, 1983, pp. 1331 - 1448.

Green80- P. E. Green, Ed., IEEE Trans. Commun., April, 1980, Vol. COM-28, No. 4, pp. 409 - 677. Later republished as Computer Network Architectures and Protocols (Plenum, New York, 1982).

Green86- P. E. Green, "Protocol conversion," IEEE Trans. Commun., March, 1986, Vol. COM-34, No. 3, pp. 257 - 268.

Groz84- R. Groz, C. Jard, and C. Lassudrie, "Attacking a complex distributed algorithm from different sides: an experience with complementary validation tools," Proc. of the Workshop on Protocol Specification, Testing, and Verification, IV, Skytop, Pennsylvania, June 1984, (North-Holland, Amsterdam, 1985), pp. 3 - 17.

Hailpern85- B. T. Hailpern, "Tools for verifying network protocols," *Logics and Models of Concurrent Systems*, K. R. Apt, Ed., NATO ASI Series, Vol. F13, (Springer Verlag, Berlin, 1985), pp. 57 - 76.

Kritzinger84a- P. Kritzinger, "Analyzing the time efficiency of a communication protocol," *Proc. Fourth Workshop on Protocol Specification, Testing, and Verification*, Skytop, Pennsylvania, June 1984, (North-Holland, Amsterdam, 1985), pp. 527 - 539.

Kritzinger84b- P. Kritzinger, "A performance model of the OSI communication architecture," IBM Zurich Research Laboratory Research Report RZ 1346, Rüschlikon, Switzerland, Dec. 11, 1984.

Krumm84- H. Krumm and O. Drobniak, "Interactive verification on the basis of CIL," *Proc. of the ACM SIGCOMM Symposium on Communications Architectures and Protocols*, Montreal, June 6-8, 1984, pp. 92 - 99.

Lam82- S. S. Lam and A. U. Shankar, "An illustration of protocol projections," C. A. Sunshine, Ed., *Proc. Second International Workshop on Protocol Specification, Testing, and Verification*, Idylwild, May 17-20, 1982, (North Holland, Amsterdam, 1982), pp. 343 - 360.

Lam86- S. S. Lam, "Protocol conversion -- correctness problems," *ACM SIGCOMM '86 Symposium*, Stowe, Vermont, Aug. 5-7, 1986, pp. 19 - 29.

Linn84- J. Linn, "An evaluation of the ICST test architecture after testing Class-4 Transport," *Proc. of the Workshop on Protocol Specification, Testing, and Verification, IV*, Skytop, Pennsylvania, June 1984, (North-Holland, Amsterdam, 1985), pp. 611 - 621.

Mills84- K. L. Mills, "Testing OSI protocols: NBS advances the state of the art," *Data Communications*, Vol. 13, No. 3, June, 1984, pp. 277 - 285.

Milner80- R. Milner, "A calculus of communicating systems," *Lecture Notes in Computer Science*, (Springer Verlag, Berlin, 1980.)

Molloy82- M. K. Molloy, "Performance analysis using stochastic Petri nets," *IEEE Trans. Computers*, Vol. C-31, No. 9, pp. 913 - 917, Sept. 1982.

Nash83- S. Nash, "Automated implementation of SNA communication protocols," *Proc. of the International Conference on Communications*, Boston, Mass., June 20-23, 1983, pp. 1316 - 1322.

Okumura86- K. Okumura, "A formal protocol conversion method," *ACM SIGCOMM '86 Symposium*, Stowe, Vermont, Aug. 5-7, 1986, pp. 30 - 37.

PDC85- Protocol Development Corporation, "Estelle development system: functional specification," Brookline, Mass., 1985.

Peterson77- J. L. Peterson, "Petri nets," *Computing Surveys*, Vol. 9, September 1977, pp. 223 - 252.

Piatkowski81- T. Piatkowski, "An engineering discipline for distributed protocol systems," *Protocol Testing - Towards Proof? (an INWG/NPL Workshop)* National Physical Laboratory, Teddington, U. K., May 27-29 1981, pp. 177 - 215.

Piatkowski83- T. Piatkowski, "Protocol engineering," *Proc. of the International Conference on Communications*, Boston, Mass., June 20-23, 1983, pp. 1328 - 1332.

Pozefsky82- Pozefsky, D. P. and F. D. Smith, "A meta-implementation for Systems Network Architecture," *IEEE Trans. Commun.*, Vol. COM-30, No. 6, June, 1982, pp. 1348 - 1355.

Rayner81- D. Rayner and R. W. S. Hale, Eds., *Protocol Testing - Towards Proof? (an INWG/NPL Workshop)*, Vol. 1: Specification and Validation, and Vol. 2: Testing and Certification, National Physical Laboratory, Teddington, U. K., May 27-29, 1981.

Rayner83- D. Rayner, "Progress in testing protocol implementations," Proc. International Conference on Communications, Boston, Mass., June 20-23, 1983, pp. 1323 - 1327.

Rayner85- D. Rayner, "Towards standardized OSI conformance tests," Proc. of the Fifth Workshop on Protocol Specification, Testing, and Verification, M. Diaz, Ed., Moissac-Toulouse, June 10-13, 1985, (North-Holland, Amsterdam, 1985), pp. 441 - 460.

Razouk84- R. R. Razouk and C. V. Phelps, "Performance analysis using timed Petri nets," Proc. Fourth Workshop on Protocol Specification, Testing, and Verification, Skytop, Pennsylvania, June 1984, (North-Holland, Amsterdam, 1985), pp. 561 - 576.

Rubin85- J. Rubin, "Testing communication protocols using random legal inputs," Proc. COMNET'85, Budapest, October 1-4, 1985, pp. 7-50 - 7-67.

Rudin82a- H. Rudin and C. H. West, "A validation technique for tightly-coupled protocols," IEEE Trans. Computers, Vol. C-31, No. 7, July, 1982, pp. 630 - 636.

Rudin82b- H. Rudin, "Validation of a token-ring protocol," in Proc. of the International Symposium on Local Computer Networks, Florence, April, 1982, (North-Holland, Amsterdam, 1982) pp. 373 - 387.

Rudin82c- H. Rudin, "Automated protocol validation: some practical examples," Proc. Sixth International Conference on Computer Communication, London, September 1982, pp. 919 - 924.

Rudin83a- H. Rudin and C. H. West, Eds., Proc. of the Third International Workshop on Protocol Specification, Testing, and Verification, Rüschlikon, May 31 - June 2, 1983, (North Holland, Amsterdam, 1983).

Rudin84- H. Rudin, "An improved algorithm for estimating protocol performance," Proc. Sixth Workshop on Protocol Specification, Testing, and Verification, Skytop, Pennsylvania, June 1984, (North-Holland, Amsterdam, 1985), pp. 515 - 525.

Rudin86b- H. Rudin, "Protocols and Complexity in Computer communications", Proc. of the 1986 International Zurich Seminar on Digital Communications, Zurich, 11-13 March, 1986, (Verlag der Fachvereine an den Schweizerischen Hochschulen und Techniken, Zurich, 1986), pp. 155 - 161.

Rudin86d- H. Rudin, "The dimension of time in protocol specification," Proc. of the IBM Europe Institute 1986: Networking in Open Systems, Oberlech, Austria, August 18-22, 1986, (to be published by Springer Verlag.)

Sabnani85- K. Sabnani, P. Wolper, and A. Lapone, "An algorithmic technique for protocol verification," Proc. GLOBECOM85, New Orleans, La., Dec. 2-5, 1985, pp. 3.7.1 - 3.7.7.

Schultz80- G. D. Schultz, D. B. Rose, C. H. West, and J. P. Gray, "Executable description and validation of SNA," IEEE Trans. Commun., Vol. COM-28, Apr. 1980, pp. 661 - 677, also reprinted in P. E. Green, Ed., Computer Network Architectures and Protocols, (Plenum, New York, 1982), pp. 671 - 705.

Shankar82- A. U. Shankar and S. S. Lam, "On time-dependent communication protocols and their projections," C. A. Sunshine, Ed., Proc. Second International Workshop on Protocol Specification, Testing, and Verification, Idylwild, May 17-20, 1982, (North Holland, Amsterdam, 1982), pp. 215 - 235.

Sherman82- M. Sherman and H. Rudin, "Using automated validation techniques to detect lockups in packet-switched networks," IEEE Trans. Commun., Vol. COM-30, No. 7, July, 1982, pp. 1762 - 1767.

Smith83- F. D. Smith and C. H. West, "Technologies for network architecture and implementation," IBM J. Res. Develop., Vol 27, No. 1, January 1983, pp. 68 - 78.

Sunshine81- C. A. Sunshine, Ed., *Communication Protocol Modeling*, (Artech House, Dedham, Mass., 1981).

Sunshine82- C. A. Sunshine, Ed., *Proc. of the Second International Workshop on Protocol Specification, Testing, and Verification*, Idylwild, May 17-20, 1982, (North Holland, Amsterdam, 1982).

Sunshine83- C. A. Sunshine, "Experience with automated protocol verification," *Proc. of the International Conference on Communications*, Boston, Mass., June 20-23, 1983, pp. 1306 - 1310.

Tenney83- R. Tenney, "Status of the ISO ad hoc Subgroup B on formal description techniques," *Proc. of the International Conference on Communications*, Boston, Mass., June 20-23, 1983, pp. 1296 - 1300.

Venkatraman85- R. C. Venkatraman and T. F. Piatkowski, "A formal comparison of formal protocol specification techniques," *Proc. of the Fifth Workshop on Protocol Specification, Testing, and Verification*, M. Diaz, Ed., Moissac-Toulouse, June 10-13, 1985, (North-Holland, Amsterdam, 1985), pp. 401 - 420.

Vissers83- C. A. Vissers, R. L. Tenney, and G. V. Bochmann, "Formal description techniques," special issue on OSI, *Proc. IEEE*, Vol. 71, No. 12, December, 1983, pp. 1356 - 1364.

West78a- C. H. West and P. Zafiropulo, "Automated validation of a communications protocol: the CCITT X.21 recommendation," *IBM J. Res. Develop.*, Vol. 22, Jan. 1978, pp. 60 - 71.

West78b- C. H. West, "General technique for communications protocol validation," *IBM J. Res. Develop.*, Vol. 22, July 1978, pp. 393 - 404.

West86- C. H. West, "Protocol validation by random state exploration," *Proc. Workshop on Protocol Specification, Testing, and Verification*, VI, Gray Rocks, Quebec, June 10-13, 1986, (to be published by North-Holland, Amsterdam.)

Yemini84- Y. Yemini, Ed., *Proc. of the Fourth International Workshop on Protocol Specification, Testing, and Verification*, Skytop, June, 1984, (North Holland, Amsterdam, 1985).

Zafiropulo80- P. Zafiropulo, C. H. West, H. Rudin, D. D. Cowan, and D. Brand, "Towards analyzing and synthesizing protocols," *IEEE Trans. Commun.*, Vol. COM-28, April, 1980, pp. 651 - 660. Also reprinted in P. E. Green, Ed., *Computer Network Architectures and Protocols* (Plenum, New York, 1982), pp. 645 - 669.

Zimmermann80- H. Zimmermann, "OSI Reference Model-- the ISO model of architecture for Open Systems Interconnection," *IEEE Trans. Commun.*, Vol. COM-28, April, 1980, pp. 425 - 432. Also reprinted in P. E. Green, Ed., *Computer Network Architectures and Protocols*, (Plenum, New York, 1982), pp. 33 - 54.

# Functional programming languages as a software engineering tool

Simon L Peyton Jones, University College London

29th August 1986

## 1. Introduction

In 1968, Dijkstra's famous letter [Dijks68] suggested that the **GOTO** statement was a harmful feature of programming languages. In due course, this gave rise to a new class of **structured programming languages**, which lacked the **GOTO** statement, but which in compensation provided a set of control structures.

In 1977, Backus' Turing award lecture [Back78] suggested that the **assignment** statement was in fact the cause of the trouble, and the **GOTO** statement was entirely innocent! This view (which has much earlier origins; see for example Landin's paper [Land66]) has given rise to the class of **functional programming languages**, which lack the notion of assignment (and hence side-effects), but which in compensation support **functions as first class citizens** (that is, functions may be passed as arguments to functions, returned as results, stored in data structures and so on).

Since then a considerable amount of work has been done on the implementation of functional programming languages, but rather less has been written about their suitability as a software engineering tool. A programming language is, of course, only one of the tools in a software engineer's toolkit, but the language chosen has a far reaching effect (for good or ill) on other aspects of the program development process. It is the purpose of this paper to suggest that functional languages are an appropriate tool for supporting the activity of programming in the large, and to present a justification of this claim.

We begin with a discussion of what we are looking for in a programming language, followed by a short introduction to functional programming (Section 3). This is followed by a discussion of the major features of functional programming which support software engineering. In particular we discuss polymorphic typing (Section 4), higher order functions and lazy evaluation (Section 5), abstract data types and modules (Section 6), formal methods applicable to functional programs (Section 7), and parallel functional programming (Section 8). Finally in Section 9 we discuss some of the major shortcomings of the functional style.

## 2. What do we look for in a programming language?

Before we can evaluate the effectiveness of functional languages we must identify the characteristics which we desire in a programming language. As Abelson and Sussman remark [Abels85],

"a powerful programming language is more than just a means for instructing a computer to perform tasks - the language also serves as a framework within which we organise our ideas about processes".

This challenge of unambiguously expressing our ideas about processes is one of the major objectives of computer science.

It is hard to improve on John Locke's remarkable insight, written in 1690:

The acts of the mind, wherein it exerts its power over simple ideas, are chiefly these three:

- (i) Combining several simple ideas into one compound one, and thus all complex ideas are made.
- (ii) The second is bringing two ideas, whether simple or complex, together, and setting them by one another so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of relations.
- (iii) The third is separating them from all other ideas that accompany them in their real existence: this is called abstraction, and thus all its general ideas are made.

John Locke, An Essay Concerning Human Understanding (1690).

The first and third of his insights apply directly to programming languages. To be specific, we look for the following characteristics in a programming language:

- (i) It should be **secure** against accidental inconsistencies.
- (ii) It should have powerful **means of combination**, by which compound expressions are built from simpler ones (Locke's first point). Examples include the control structures of structured programming, data structures and so on.

Powerful combining forms lead to **concise** programs, so that we can hold the whole program in mind at once and "think big thoughts all at once". They also lead to **modular** programs, which are built up systematically out of smaller pieces, rather than being a single monolithic structure.

- (iii) It should have powerful **means of abstraction**, by which compound objects can be named and manipulated as units (Locke's third point). Examples include function and procedure definition, modules, classes and so on.

Powerful means of abstraction also support **modular** programming.

- (iv) It should be **amenable to formal methods** of program specification, reasoning, transformation and analysis.
- (v) With the advent of parallel machines it is highly desirable that the language should be able to express **parallel computations**.
- (vi) It should be possible to construct an **efficient implementation**, since we want to actually run our programs. Furthermore it should be possible to **reason about the performance and resource requirements** of a particular program.

This list of characteristics gives the structure of the rest of this paper; we will discuss them in the order they are given above. First, however, we offer an introduction to functional programming, which may safely be omitted by the cognoscenti.

### 3. A brief introduction to functional programming

In this section we will give a very brief introduction to functional programming. Other introductions to functional programming are given by Turner [Turn82] and Darlington [Dar184]. For fuller treatments see Burge [Burge75], Henderson [Hend80], and Glaser et al [Glas84]. The best book about using the functional idiom as a means of writing good programs is undoubtedly Abelson and Sussman's excellent book [Abels85].

Functional languages are members of the class of **declarative languages**, that is, languages in which programs have a **declarative reading** which asserts properties of the functions defined by the program, as well as an **algorithmic reading** which describes how the functions are to be computed. Another common term for functional languages is **applicative languages**. By way of contrast with the declarative nature of functional languages we will refer to conventional procedural languages (like Pascal or C) as **imperative**.

For the sake of definiteness, examples given in this paper will be written in the language Miranda [Turn85]. However Miranda is by no means the only functional languages, and we summarise the main ones at the end of this section.

### 3.1 Definitions and scripts

A Miranda program consists of a set of definitions (the **script**) together with an expression to be evaluated. Executing the program consists of evaluating the expression and printing the result. For example, the empty script together with the expression:

```
3+4
```

is a valid Miranda program. When executed it will print the result:

```
7
```

Here is a possible Miranda script:

```
a = 4
b = 5
c = a-b
```

Each line is a separate definition, and the script gives values to the identifiers "a", "b" and "c". These definitions are closely analogous to the Pascal **const** declaration, which gives a value to an identifier, which remains constant throughout its scope. They are quite different to assignment statements, which change the value of an identifier. The identifiers are names for values, not names for boxes containing values.

With this script, the expression:

```
b-c
```

would evaluate to:

```
6
```

Functions may be defined in the script. For example the script:

```
average x y = (x+y)/2
```

defines "average" to be a function which takes the mean of its two arguments. The expression:

```
average 4 6
```

evaluates to:

```
5
```

Notice that function application is denoted by juxtaposition, so that we write "average 4 6" not "average(4,6)". Parentheses are used solely for grouping, rather than having two purposes as they do in Pascal.

Identifiers can also be bound to values locally, using a "where" clause:

```
f x = y+1 where y = 2*x
```

Functions can be defined locally in a similar manner.

Miranda is a strongly typed language, and the compiler deduces a type for each identifier (see Section 4). The type of a factorial function, for example, would be

```
factorial :: num -> num
```

That is, factorial is a function which takes a number and returns a number. The type of average is

```
average :: num -> (num -> num)
```

That is, "average" is a function which takes a number and returns a function, which takes a number and returns a number. Thus, we may form a **partial application** of "average"; for example, the expression

```
(average 4)
```

is the function which averages its argument with 4. This way of treating a function of several arguments as a function of a single argument which returns a functional result is called **currying**.

### 3.2 Guards and pattern matching

Two other constructs aid in function definition, **guards** and **pattern matching**. An example of a function using a guard is the greatest common divisor function:

```
gcd a b = gcd a (b-a),    a>b
         = gcd (a-b) b,    a<b
         = a,              a=b
```

This definition is reminiscent of a mathematician's syntax, and gives three alternative right hand sides for "gcd". The appropriate one is chosen depending on the value of the guard, which is a boolean expression following a comma on the right hand side. The last guard may be omitted to give an "otherwise" case.

An example of a function using pattern matching is:

```
factorial 0 = 1
factorial n = n * (factorial (n-1))
```

The appropriate equation is chosen depending on the value of the argument, by matching it against the pattern on the left hand side. In this case we could use a guard instead, but pattern matching comes into its own when data structures are introduced.

### 3.3 Data constructors and types

The major built in data structure is the list. For example, the expression:

```
[4,3,9]
```

denotes a list of three numbers. Lists may be built with an infix list construction operator ":". Thus the expression:

```
4:[3,9]
```

denotes a list whose first element is 4 and the rest of which is [3,9]; it is the same list as [4,3,9]. The primary way in which lists are taken apart is by using pattern matching. For example, the function which takes the length of a list could be defined like this:

```
length [] = 0
length (x:xs) = 1 + (length xs)
```

(Note: the parentheses around "length xs" are not required.) The empty list is denoted [], so the first equation says that the length of the empty list is 0. The second equation says that the length of a list whose first element is "x" is 1 plus the length of the rest of the list.

Miranda allows the programmer to declare new types. For example, the script

```
tree ::= EMPTY | NODE tree num tree
```

defines a new type "tree", and says that an object of type "tree" is

- either EMPTY,
- or a NODE with three fields, of type "tree", "num" and "tree" respectively.

Objects of type "tree" can be built using the constructors EMPTY and NODE; for example

```
EMPTY
NODE EMPTY 3 EMPTY
NODE (NODE EMPTY 4 EMPTY) 5 EMPTY
```

are all trees. Trees can be taken apart using pattern matching. For example, the function "sum" adds up all the numbers in the tree:

```
sumtree EMPTY           = 0
sumtree (NODE t1 n t2) = sumtree t1 + n + sumtree t2
```

### 3.4 Other functional languages

The language ML, originally designed as part of the Edinburgh LCF project [Gord79], is now being significantly redesigned to take account of the experience gained with ML and other functional programming languages [Miln85]; the new language will be called Standard ML. A version of ML with lazy semantics has been implemented by Johnsson and Augustsson at the Chalmers University of Technology [Augus84].

Hope, designed by Burstall, MacQueen and Sannella [Burst81] at Edinburgh, is the language which Darlington's team at Imperial College are using for the ALICE project. It is a large language (for instance, it supports user-defined operators which may be prefix, infix or distfix).

Turner has designed a series of languages, Sasl [Turn76], KRC [Turn82] [Hamm84], and now Miranda.

Ponder is a language designed by Fairbairn with an interesting generalisation of the ML type system [Fairb82], while Cardelli's Amber language generalises ML's types in a different way [Card85].

SISAL, a functional language originating from the dataflow stable and targeted particularly at numerical computation, is in the late stages of development by a partnership between the University of Manchester, MIT, DEC and Lawrence Livermore National Laboratory [Univ85].

Lispkit, designed by Henderson [Hend80], is a functional language which shares the syntax and simplicity of Lisp, and is explicitly designed for easy portability.

Despite this plethora of languages, there are many issues where a consensus is emerging, and it is this consensus that we attempt to present here.

### 4. Polymorphic typing

Strong typing has been widely accepted in conventional programming practice as a technique which catches a large class of programming errors at the compilation stage. Unfortunately, strong typing sometimes leads to tiresome bureaucracy, and this problem becomes particularly serious in a functional programming style.

Consider, for example, the "length" function defined earlier. The function will work equally well on a list of numbers and on a list of characters, but a conventional strong typing system will force the programmer to write separate functions for these two data types.

The ability to write such generic functions is so important to functional programming that early functional programming languages were typeless. Fortunately, however, based on the work of Hindley [Hindl69], Milner developed the concept of polymorphic typing [Miln78] [Damas82], which generalises the notion of type to allow types such as list of \*, meaning

"a list of objects of any type". Now the length function has a well behaved type:

```
length :: [*] -> num
```

(Miranda uses "[\*]" to mean "list of \*".) Likewise, a function which reverses a list has type:

```
reverse :: [*] -> [*]
```

Milner also showed that the type of an expression can be **inferred from the source program by the compiler**, rather than explicitly declared by the programmer. This is the approach taken in ML, the first language to include polymorphic typing; Cardelli describes the method [Card85a].

Miranda also infers its types, but allows optional type declarations; if they are present, the compiler will check them [Turn85a].

Polymorphic typing thus gives us the best of both worlds - we can have complete type checking of fully generic programs, without having to write any explicit type declarations.

Type inference turns out to be quite a delicate balance between the expressiveness of the language and the computability of the type checking algorithm. For example, Hope also includes a polymorphic typing system, but it also allows **overloading** of operators, and this turns out to require explicit type declarations.

Even more general polymorphic type systems are exemplified by the languages Ponder [Fairb85] and Amber [Card85], both of which also require explicit type declarations.

Polymorphism is not restricted to functional languages, and it is applicable to both conventional imperative languages [Harl84] and logic languages [Mycr84]. However, functional languages seem to have provided a particularly hospitable framework for polymorphic typing systems.

The best overview of the world of types and polymorphism is given in a recent paper by Cardelli and Wegner [Card85b].

## 5. Means of combination

In common with imperative languages, functional languages provide many ways of "gluing together" simple expressions to form more complex ones. Powerful glue leads to

- (i) **Concise programs.** It is well established that textually short programs can be written faster and debugged more quickly than long programs (see for example Chapter 8 of Brooks [Brook82]). Concise programs allow us to "think big thoughts in one go". The value placed on conciseness can be seen by the fact that "very high level programming language" means "a language in which we can write big programs with few symbols".

This approach is not without its dangers, however (vide APL).

- (ii) **Modular programs.** For a software engineer modularity is one of the key properties of a language. A modular program allows the programmer to restrict his attention to one part of the program to the exclusion of the rest. The availability of powerful glue is one of the prerequisites to modular programming, since the modules need to be "stuck together".

Some of the combining forms of functional languages are very similar to those of any other language; for example

- Function application.
- Local definitions using "where".
- Data constructors (often user-definable).

In addition, functional languages provide two forms of glue which are almost unique, namely **higher order functions** and **lazy evaluation**. These two features are explored in detail by Hughes [Hugh84]. A more recent innovation is the possibility of incorporating unification in functional language, a concept which we explain below.

We will discuss these ideas in turn, but first we mention two notational devices which have proved to be highly successful glue.

### 5.1 Pattern matching

The use of pattern matching to perform case analysis implicitly rather than using explicit conditionals has gained wide acceptance. All recently designed functional languages incorporate pattern matching in roughly the form outlined above.

The use of pattern matching leads to a very clear program structure, in which a function taking an argument of type "thing" will normally contain an equation for each possible "thing". Thus the function "sumtree" defined above contained one case for each kind of tree.

### 5.2 List comprehensions

Consider this program, which computes the infinite list of prime numbers:

```
primes = sieve [2..]
sieve (p:x) = p : sieve [ n | n <- x; n rem p > 0 ]
```

where [2..] denotes the infinite list of integers beginning with 2, "<-" means "drawn from", and "rem" is the remainder operator. The expression after "p : sieve" is called a **list comprehension**, and can now be read as "the list of all n, drawn from (the list) x, such that the remainder when n is divided by p is greater than zero".

This notation for describing a list, inspired by Zermelo-Frankel set notation, was first introduced in a functional language by Burstall and Darlington in NPL [Burst77], and taken up by KRC, Hope and Miranda. It is a declarative construct, and fits very gracefully into a functional context. There seems to be no doubt that it makes functional programs significantly more concise, although its acceptance is not yet as widespread as pattern matching.

### 5.3 Higher order functions

In functional languages, a function is a "first class citizen". It may be treated as an object in the same way as any other data object, passed to another function as an argument, returned as a result, stored in a data structure, and so on. A higher order function is a function which takes a function as an argument and/or returns a function as a result.

#### 5.3.1 An example

Borrowing an example from Hughes' paper, consider the function which adds up the elements of a list of numbers:

```
sumlist [] = 0
sumlist (x:xs) = x + sumlist xs
```

The only parts of this function definition which are specific to adding up the elements of a list (rather than, say, multiplying them together) are the "0" of the first equation and the "+" of the second. Suppose we parameterised the definition of the "sumlist" function on these two values, giving a new function "fold", where we intend that

```
newsumlist = fold (+) 0
```

(Note: we have to enclose the "+" in brackets before passing it to "fold" to avoid the parser interpreting "fold + 0" as the sum of "fold" and 0! This is just another example of using brackets for grouping.) The definition says that "newsumlist" is a function obtained by applying the general function "fold" to "+" and 0. We can obtain the definition of "fold" by a simple generalisation of the definition of "sumlist":

```
fold binop unit [] = unit
fold binop unit (x:xs) = binop x (fold binop unit xs)
```

(Note: we have to use prefix notation for "binop" because only built in operators like "+" are parsed as infix.) Notice that while "fold" is a function expecting three arguments, we can **partially apply** it to only two arguments in the definition of "newsумlist" (see Section 3.1). Having defined "fold" in this way, we can now define numerous other useful functions:

```
productlist = fold (*) 1
alltrue = fold (&) true
anytrue = fold (|) false
```

"productlist" multiplies the elements of a list of numbers together, "alltrue" and's together a list of booleans, while "anytrue" or's them together.

Another way to understand what "fold (+) 0" does is to observe that it takes a list such as:

```
3 : 5 : 34 : []
```

(":" associates to the right), and returns the expression:

```
3 + 5 + 34 + 0
```

In other words, the ":" has been replaced by "+" and the "[]" has been replaced by 0. This can also be seen by referring to the definition of "fold". We can now see that:

```
fold (:) [] xs
```

will simply return a copy of the list "xs", since we replace ":" with ":" and "[]" with "[]". Similarly, if "ys" is a list, then

```
fold (:) ys xs
```

will return a copy of the list "xs", except that the "[]" at the end will be replaced with the list "ys". Thus

```
append xs ys = fold (:) ys xs
```

defines the function "append" which concatenates two lists. Finally, for the stout-hearted, we can even use "fold" to define "map", where "map f xs" applies the function "f" to each element of the list "xs":

```
map f = fold ((:) . f)
```

where the infix "." denotes function composition, so that "(f . g) x" evaluates to "f (g x)".

### 5.3.2 Conclusions

The function "fold" is a higher order function, since it takes a function as an argument. The ability to define higher order functions allowed us to generalise our original definition of "sumlist", so that instead of a monolithic definition of "sumlist" we were able to build it up out of the modules "+" and 0 using "fold" as glue. Of course this is a rather small example, but we were nevertheless able to derive quite a rich variety of functions by using "fold" as a means of combination of other simple functions.

Higher order functions are therefore rather like the control structures of conventional languages. They combine simpler functions together to form more complex ones. Unlike conventional languages, however, which have a fixed set of control structures, a functional language allows new higher order functions to be defined at will.

In summary, it is as though the language allowed new control structures to be defined whenever desired, to precisely fit the application.

## 5.4 Lazy evaluation

One mechanism for gluing programs together which the Unix\* operating system provides is the **pipe**, which allows the output of one program to be fed directly into another. For example, the command:

```
tbl paper | eqn | nroff
```

feeds the file "paper" into the "tbl" program, sending the output directly into the input of the program "eqn", whose output is fed directly to the program "nroff". Unless the file "paper" is very small then, rather than completely process "paper" with "tbl" and writing the output to a temporary file before invoking "eqn", Unix will instead run "tbl", "eqn" and "nroff" all at once, with moderate sized buffers in between each. This avoids the need for large temporary files, and makes the first output appear as early as possible.

The popularity of Unix may partly be attributed to the ease with which programs can be glued together in this way. Some functional languages provide a rather similar sort of glue in a much more thorough-going way, through a technique called **lazy evaluation**. Suppose the result of a function "f" is a large list, which is passed to another function "g":

```
g (f x)
```

Then we would prefer that the list produced by "f" is only generated as "g" uses it up, and this is just what lazy evaluation achieves. In effect "f" and "g" run as coroutines; "g" runs until it needs the next item of the list, whereupon "f" is awakened. It runs until it has produced the next item, and then goes back to sleep to allow "g" to continue. We can think of "g" **demanding** the next list item from "f", and this gives rise to the term **demand driven evaluation**.

In a lazy functional language all data structures are only evaluated on demand. Just as in the case of Unix, lazy evaluation provides powerful glue for combining programs together. Unlike Unix, however, which requires a special construct to express this combining form, simple function composition serves to combine lazy functions together.

Lazy evaluation has a far reaching effect on program style. Consider, for example, a function which computes square roots by iterative approximation. The formation of successive approximations and the termination condition are intimately interwoven in an imperative language. In a lazy functional language we can separate the square root function into two functions, one of which generates the (infinite) list of all approximations to the square root, and the second of which eats up this list until it finds a sufficiently good approximation. This is a more modular approach, since we can now replace the first function with a superior method of generating approximations without affecting the second (and vice versa).

Lazy evaluation has allowed us to modularise the square root function into two independent sub-functions, by taking the termination condition out of the loop body. This is a typical of the way in which lazy evaluation aids modularity; the program is factored into a generator function which constructs a number of possible solutions (the approximations in the square root case) and a selector function which examines them in turn and chooses the appropriate one. A more sophisticated numerical example is given by an arbitrary precision arithmetic package, where the precision of the answer can be arbitrarily increased in a demand driven manner [Peyt83].

As another example, consider a chess-playing program which works by exploring the tree of possible moves. We could write the function which evaluates the value of a position "p" like this:

```
minimax (prune (generate_moves p))
```

The function "generate\_moves" generates the (possibly infinite) tree of moves starting at position "p", while "prune" prunes the tree in some suitable way to make the tree small enough to apply "minimax" which performs the well known mini-max algorithm. Lazy evaluation ensures that the tree will only be generated as it is used. The program is modular, so that, for example, we could change to using the alpha-beta heuristic instead of mini-max without changing "prune" or "generate\_moves". In an imperative language these three functions have to be inextricably tied up together in a single monolithic structure.

## 5.5 Adding unification to functional languages

Logic languages such as Prolog [Clock81] are also free from side effects and hence declarative [Kowal79] (so long as we eschew Prolog's dynamic ASSERT, which causes a side effect). However, the relationship between functional and logic languages is still extremely unclear.

Logic languages are based on the extremely powerful **unification** operation [Robin71] (a sort of two-way pattern match, by contrast with the one-way pattern matching of functional languages), which, among other things supports **backtracking** and, in principle, allows programs to be run **backwards** as well as forwards (though running the program backwards is normally too inefficient to be feasible). Logic languages are only **first order**, and lack the power of higher order programming.

Unification is, however, clearly an extremely powerful glue for building programs, and it is natural to ask whether it can be incorporated into the framework of functional languages. In 1983 Darlington suggested a method of extending functional languages to include unification [Darl83], by extending the **relative** list comprehension described above to **absolute** list comprehension. Then, instead of writing

$$[x \mid x \leftarrow L; p \ x]$$

meaning "the set of  $x$  drawn from  $L$  such that  $(p \ x)$ ", we might write

$$[x \mid p \ x]$$

meaning "the set of  $x$  drawn from  $S$  such that  $(p \ x)$ , where  $S$  is the set of all values to which  $p$  is applicable" (the applicability of  $p$  is determined by the type-checker). As an example, consider

$$\text{split } s = [ [s1, s2] \mid \text{append } [s1, s2] = s ]$$

where **append** is defined by

$$\begin{aligned} \text{append } [], k2 &= k2 & (1) \\ \text{append } [x:k1], k2 &= x : \text{append } [k1, k2] & (2) \end{aligned}$$

Now suppose  $s = [1, 2]$  (so that  $s = 1:[2]$ ). We have

$$\text{split } [1,2] = [ [s1, s2] \mid \text{append}[s1, s2] = 1:[2] ]$$

and now we must perform a unification with the definition of **append**, to get (as one possibility, by unifying with (2))

$$\text{split } [1,2] = [ [1:k1, k2] \mid \text{append } [k1, k2] = [2] ]$$

Taking it one step further, unifying with (1)

$$\text{split } [1,2] = [ [ [1], [2] ] ]$$

which is one valid result. The others will be generated from choosing different equations with which to unify.

Darlington shows that the technique will work for higher order functions as well, but there is a limit here, since unification is known to be undecidable at second order and above [Huet73] [Huet75] [Goldf81] [Pages83].

There has been a recent flurry of papers on the subject (see for example [Robin84] [Reddy85] [Linds85] [Darl85]), and there seems no doubt that the idea is here to stay.

## 6. Means of abstraction

Suppose a manufacturer wishes to design a new car. First of all specifications will be established for the various components (brakes, engine, audio hifi system, and so on), and then the design of these components will be assigned to different individuals. It is taken for granted that the designer of the braking system does not need to know anything about the design of the hifi system, because these subsystems do not interact.

Modularity of this kind is perhaps our primary tool for managing complexity. Unfortunately, essential though it is, modularity cannot be taken for granted in program design. It is all too easy for unforeseen interactions between program modules to cause the system to fail. The programming language should therefore provide mechanisms to **control the interactions** between different parts. All languages do this to some extent, but functional languages are particularly strong in this respect.

### 6.1 Abstraction through function definition

The simplest form of abstraction, which is supported by any programming language worthy of mention, is functional (or imperative) abstraction. The function:

```
append xs ys = fold (:) ys xs
```

abstracts the idea of concatenating two lists away from the method of concatenation. We are free to change the implementation of the function so long as we do not change the abstraction it implements. At the most trivial level we are free to change the name of the formal parameters, but we could make a more significant change, by writing an explicit definition of append:

```
append [] ys      = ys
append (x:xs) ys = x : append xs ys
```

Functional languages enforce a very clearly defined interface between a function and its caller. The result of a function is governed solely by its arguments, and if the same function is applied to the same arguments on a different occasion the same result will be produced. This can be guaranteed because the absence of assignment means that no side effects can take place between the two calls.

In a large Pascal program it generally proves impossible to communicate between functions entirely by parameter passing, so some communication is mediated by shared global variables which are modified as a side effect of calling the function (or procedure). This allows a potentially undisciplined interaction between functions, and it is often an unexpected side effect which causes an imperative program to fail. Modularity is lost, because the insides of one module interact in an intimate way with the insides of another module. The alternative of restricting imperative programs to eschew all side effects (except those local to a single function) proves too restrictive, because imperative languages have insufficiently powerful combining forms and storage allocation mechanisms.

Functions in a functional language implement the mathematical abstraction of a function, whereas functions in an imperative language implement something altogether more complicated and harder to understand.

### 6.2 Abstract data types and modules

Many modern imperative languages, such as Ada\*, contain mechanisms to support the development of large systems. Examples of such mechanisms include modules, packages or classes. They constitute the next level of our attempt to manage complexity, by breaking the system into components of manageable size which can then be combined together in a secure way to form a complete system. Typically they support abstract data types, multi-tasking, and generic shareable program components. They all implement information-hiding in some form, to enforce limitations on the forms of interaction between components.

Recent work strongly suggests that functional languages form a hospitable environment for such efforts. ML successfully supported abstract data types from its inception [Gord79], using a mechanism supported almost entirely by the type checker, imposing no run time overhead. Miranda provides an even simpler mechanism with similar properties [Turn85a]. As an example of the power of these mechanisms, the LCF system written in ML has an abstract type "theorem". Objects of this type are guaranteed to be theorems of the logic concerned because such objects can only be produced by combining together other theorems and axioms using the laws of deduction provided by the abstract type.

Attention is now turning to more sophisticated constructs. In imperative languages the mechanisms for constructing generic packages, for example, tend to be rather complex and ad hoc. Recent work in typing systems for functional languages suggests that it will be possible to make modules "first class citizens", capable of being typed, named, parameterised and processed. Just as treating functions as "first class citizens" opened the way to higher order functions, and user-defined combining forms, so this treatment of

modules will allow the programmer to instantiate and combine modules using combining forms defined for the purpose.

This work is at an experimental stage at present, but the signs look encouraging. The most accessible points at which to begin further reading are Burstall and Lampson [Burst84] [Burst84a], MacQueen [MacQ85], and Cardelli and Wegner [Card85b].

## 7. Formal methods

The use of formal methods to reason about programs is the very stuff of software engineering. Formal methods are a key tool in the transformation of programming from black art into an engineering discipline. Our aim is to generate efficient and correct programs from formal specifications, and to do so using machine-checkable techniques.

In order for such formal methods to be practicable it is essential that the programming language should have simple semantics, for otherwise the tasks of program transformation and verification become extremely burdensome. Lacking the assignment statement, functional languages do indeed have extremely simple semantics by comparison with almost any "real" imperative language. In this section we will discuss two areas where mechanised formal methods have been successfully brought to bear.

### 7.1 Program transformation

The primary goal of program transformation methods is to transform a comprehensible though inefficient **specification** into an efficient but possibly obscure **program** [Darl81]. Closely related goals are the design of efficient data representations [Darl80], and proving properties of programs [Turn83].

This is an expanding field, and a rich source of references may be found in [Parts83]. The most commonly encountered framework for program transformation is the Fold/Unfold system of Burstall and Darlington [Burst77a], and we give a short example here to give the flavour of the subject, taken from [Darl81]. Assume that the following functions are defined (we use curly brackets on the left to identify equations):

```
{lg.1} length [] = 0
{lg.2} length (x:s) = 1 + length s

{ap.1} append [] s2 = s2
{ap.2} append (x:s1) s2 = x : append s1 s2
```

and that we want to define a function to append two lists together and find the length of the resulting list. We can write it thus:

```
{lo2} lengthof2 s1 s2 = length (append s1 s2)
```

but while this definition is adequate and clear, it is somewhat inefficient, and we will derive a more efficient version. Instantiating {lo2} with s1=[] gives

```
lengthof2 [] s2
= length (append [] s2)      instantiating by {lo2}
{lo2.1} = length s2           unfolding by {ap.1}
```

Now instantiating {lo2} with s1=(x:s1) gives

```
lengthof2 (x:s1) s2
= length (append (x:s1) s2)  instantiating by {lo2}
= length (x : append s1 s2)  unfolding by {ap.2}
= 1 + length (append s1 s2)  unfolding by {lg.2}
{lo2.2} = 1 + lengthof2 s1 s2 folding by {lo2}
```

Finally, we see that {lo2.1} and {lo2.2} constitute a complete new definition of lengthof2, which is "more efficient" than the first version.

### 7.1.1 Transformation Systems

The transformation given above is completely formal, and amenable to machine **checking**. However, it is not very clear what "more efficient" means, so machine **generation** of proofs is problematic.

While in the example above the sequence of transformations applied is fairly simple, larger examples become considerably more complex, and there is no doubt that, even with machine assistance, it would become impossible to conduct large proofs one step at a time. There are two basic strategies for addressing this problem, which Partsch and Steinbruggen [Parts83] call the **generative set approach** and the **catalog approach**.

The generative set approach uses a basic set of elementary transformations (fold, unfold, instantiate and so on), together with some kind of **meta language** in which to express patterns of transformation, so that the proof can be conducted at a higher level of abstraction (cf [Gord79], where ML is used as a proof meta language, but in a different proof system). The proof then becomes a program in the meta language. Darlington is using Hope itself as a meta language for conducting proofs about Hope programs [Dar181], and this is also the approach taken by the large and impressive CIP project [Moll84].

The catalog approach works by proving, in advance, a sufficiently large set of powerful theorems about the (higher order) functions involved in the program. These theorems are, in effect, "high level transformations" and therefore much more tractable than the elementary transformations of the generative approach. The catalog approach works well when there is a very restricted set of higher order functions, such as in the language FP, where it leads to an "algebra of programs", as described by Backus [Back78]; further references are [Wad181] [Islam81] [Kieb81].

### 7.1.2 Program transformation in compilers

So far we have discussed program transformation in the context of supporting the program development process. This is an ambitious and long term goal. Program transformation has also been applied with considerable success in the more limited context of program compilation, and large sections of almost any compiler for a functional language can be regarded as program transformations. They are thus amenable to formal proofs of correctness, and are readily communicable to other compiler writers.

Examples abound, including:

- The combinator abstraction algorithm of Turner [Turn79].
- The compilation of pattern matching [Augus85] and list comprehension constructs [Wad186].
- The supercombinator (or lambda lifting) algorithm of Hughes [Hugh82] [Johns85].
- The optimisation of functional programs by partial evaluation [Hudak84] [Jones85].

## 7.2 Abstract interpretation

Abstract interpretation is a technique for deducing information about a program from its text, by executing an abstract version of the program. An appropriate abstraction is chosen according to what information is wanted.

As an example, suppose we wanted to know the sign of

$$34298 * (-3981) * 300993$$

The hard way to find the sign of this number is to perform the two multiplications in full and look at the sign of the result. A simpler method is to perform a more abstract calculation:

$$+ <*> - <*> +$$

We replace each number with an abstract representation (its sign), and replace the multiplication operator with an abstract version "<\*>", which implements the well-worn "rule of signs".

```

+ <*> + = +
- <*> + = -
+ <*> - = -
- <*> - = +

```

Now it is easy to compute the answer "-", which tells us that the result of the original calculation would have been negative. The abstraction we chose suited the question we wanted to ask, and the execution in the abstract domain was much less tedious than the original.

Abstract interpretation is an area of much recent research, since the technique seems to have potentially wide applicability. It is in one area, however, that of **strictness analysis** that most progress has been made.

### 7.2.1 Strictness analysis

Functional languages are promoted as a natural vehicle for programming parallel machines, so it is legitimate to ask where such parallelism comes from.

It turns out that it all comes from evaluating the argument(s) of a function in parallel (with each other and with the function itself). When, then, can we evaluate the argument of function in parallel? It can certainly never render our program incorrect (since it can have no side effects), which is reassuring, but it may not be a good way to use the machine's resources.

One approach is to drop the problem in the programmer's lap, and require annotations in the program to indicate where parallelism is desired. However, we might prefer to find some automatic technique for discovering when it is **safe** to evaluate an argument in parallel. In this context, it is "safe" to evaluate an argument to a function in parallel if we know that the function will need the value of its argument, and hence must eventually evaluate it. For instance, consider the function

```
f x y = if (expensive y) then x+1 else x+2
```

where "expensive" is some expensive function returning a boolean. We know that x's value will be needed, and we could evaluate it in parallel. This kind of analysis is sometimes called **strictness analysis** (a function is strict if it always evaluates its argument).

Mycroft's thesis [Mycr81] describes a method for performing this strictness analysis using abstract interpretation. Mycroft suggests reducing the data domain to "undefined" and "defined". Then if, by executing the program in this abstract domain, we find that

```
f "undefined" = "defined"
```

then f does not evaluate its argument, so we must not evaluate the argument in parallel (because it may not terminate). If, on the other hand

```
f "undefined" = "undefined"
```

then f does evaluate its argument. In this case, it is safe to evaluate the argument in parallel. Matters are in fact more complicated than this, but only slightly, and Mycroft's work comes complete with a sound theoretical basis. However, his work only applies to flat domains (ie no data structures) and first order functions.

Recent work has, however, extended the method to higher order functions [Burn85] and data structures [Hugh85] [Wad185].

## 8. Parallel computation

One of the most attractive features of functional programming languages is that they are not inherently sequential, as conventional imperative languages are. As Section 3 implied, a functional program is executed by evaluating an expression, and several sub-expressions may be evaluated concurrently. For example, in the expression

```
(3+4) * (sqrt 24)
```

the addition can take place concurrently with the square root. Thus the hope offered by functional languages is that

parallel execution of functional programs, through concurrent evaluation of sub-expressions; may be possible without adding any new language constructs or detailed program tuning.

If taken without qualification this statement is rather misleading, since it seems to promise "parallelism without tears", but in fact co-operation never comes for free. We can, however, take the statement as highlighting an opportunity. If the challenge of parallel machines is a mountain then concurrent graph reduction offers us a promising looking path between the steep cliffs.

The idea of concurrent execution of programs without adding new language constructs is not new. The FORTRAN compiler for the Cray-1 is designed to spot vectorisable sections of programs written in (almost) ordinary FORTRAN. However, the effective use of the Cray relies on the programmer writing his program in such a way that the compiler can spot that it is vectorisable. Furthermore, high performance is an extremely delicate property of programs, and easily destroyed by seemingly innocuous modifications.

We hope that in the case of functional languages the parallelism is less delicate and more general, so that the programmer's task is made easier. First, therefore, we will discuss the task of writing parallel functional programs.

### 8.1 Writing parallel functional programs

It is tempting to believe that an arbitrary functional program would run much faster on a parallel graph reduction machine. This comforting belief is quite erroneous [Clack85]. Many functional programs are essentially sequential (that is, at any moment there are few redexes in the graph). For example, an insertion sort program cannot insert the next element into the result until the previous insertion has completed (or at least partly completed). It is simply unreasonable to expect any old functional program to run fast on a parallel machine.

In order to achieve good parallel performance the program must contain **algorithmic parallelism**. That is, the algorithm must contain gross inherent parallelism. The most obvious sort of algorithmic parallelism is given by **divide and conquer** algorithms, which divide the task at hand into two or more independent sub-tasks, solve these independently, and then combine the results to solve the original task. A standard example of such an algorithm is quicksort, which splits the set to be sorted into two subsets which can be sorted independently. Other examples include any kind of search algorithm (which covers many Artificial Intelligence applications) and large numerical computations. Experiments confirm that substantial parallelism is obtainable [Tighe85] [Clack85].

It is therefore still the program designer's responsibility to create an algorithm which will divide the task at hand into reasonably independent sub-tasks. It is unreasonable to expect the machine to do this automatically, since it may involve major algorithmic changes (such as changing insertion sort to quicksort).

### 8.2 Writing parallel programs is easier in a functional language

Why not program in a conventional language which supports multiple tasks, such as Ada? There are a number of ways in which writing a parallel program in a functional language is superior to this:

- (i) In conventional languages the division of the problem into separate tasks is static and fixed. A task is conceived as a relatively large unit, and tasks generally cannot be created and deleted dynamically. There will be relatively few tasks, and the programmer must clearly identify all of them in his design.

In a functional language the parallelism can be dynamic, and there is no static division of the problem into tasks. Instead, the programmer designs an algorithm whose inherent parallelism will enable concurrent reduction to take place at different places in the graph. The "grain" of parallelism is therefore smaller and more dynamically adaptable as the computation proceeds.

- (ii) In conventional languages the tasks communicate with each other by sending messages or making specially protected subroutine calls to each other. The programmer has to design synchronisation and communication protocols between tasks so that they cooperate correctly and achieve mutual exclusion where necessary. It is up to the programmer to ensure that these communication protocols are correct, and failure to do so can result in a transient malfunction of the program.

In a functional program the synchronisation between different reductions is mediated entirely by the shared graph. A reduction is made known to the graph by the indivisible operation of overwriting the root of the redex with the result of the reduction, and no other synchronisation is necessary (though see the next section for efficiency considerations).

- (iii) The tasking structure of conventional languages adds a layer of considerable complexity to the programmer's model of what is going on. If it is hard to reason about a sequential program, it is even harder to reason about a multi-tasking program, because the programmer has to bear in mind all the possible time orderings in which execution might take place. The behaviour of the program should be independent of the scheduling of the tasks, but it is up to the programmer to ensure that this is the case.

There are no extra language constructs required to write parallel functional programs. The result of the program is guaranteed to be independent of the way in which reductions are scheduled, though this scheduling may have a strong impact on efficiency. Thus it is no harder to reason about a parallel functional program than a sequential one.

To summarise, the programmer does not have to design a static task structure, guarantee mutual exclusion and synchronisation, or establish communication protocols between tasks. This frees him for the creative activity of designing a parallel algorithm.

## 9. Shortcomings of functional programming

It would be misleading to present functional languages as the panacea for all ills, and this section summarises their principal shortcomings.

Until recently the raw efficiency of functional languages has been a problem, but recent implementations now approach the speed of compiled Pascal [Augus84a] [Fairb85a], so the compiler technology seems now available to alleviate this problem.

### 9.1 Reasoning about performance

The goal of a programmer is to write programs that are

- (i) (absolutely) correct; that is, they should meet their specification.
- (ii) (reasonably) efficient; that is, they should consume as few machine resources as possible.

In order to achieve these goals the programmer has to reason about

- (i) the meaning of his program, to assure himself that it has the same meaning as the specification.
- (ii) the resource consumption of his program, to assure himself that it will consume only reasonable resources.

We have argued that functional languages are much more tractable than imperative ones when reasoning about correctness. On the other hand, it is normally fairly straightforward to reason about the memory space and CPU cycles consumed by a imperative program, because the programmer has an accurate mental model of how execution takes place.

Unfortunately, a major weakness of functional languages is the difficulty of reasoning about their space and time behaviour, especially the former. In particular, a functional program will often have much worse space-time behaviour than the programmer might expect. The trouble is that it is difficult for the programmer to maintain an accurate mental model of how execution takes place, and whatever understanding he may develop is normally

wrecked by the implicit coroutine behaviour of lazy evaluation. Furthermore, seemingly innocuous (and correctness preserving) changes to the program may have truly dramatic effects on its run-time behaviour. For example, the space required by a three line program to compute the set of all sublists of a list of length  $N$  can change from  $O(N)$  to  $O(N!)$  as a result of a minor program change (common subexpression elimination).

Meira takes the efficiency of functional programs as the main subject of his thesis [Meira85], and chapter 6 of Stoye's thesis [Stoye85] gives a good summary of the various guises in which this problem manifests itself.

The unpredictable performance of functional programs is possibly the most serious obstacle to their widespread adoption in production environments.

## 9.2 Input/output and nondeterminism

The absence of side effects in functional programs raises the question of how input and output is to be performed. The solution is to regard the program as a function mapping the input data (from the user's keyboard or a file) to the output data (which the operating system can send to the user's screen or a file).

```

-----
Keyboard ----> | Functional program | ----> Screen
-----

```

Lazy evaluation is a big help in this context, because it means that the (potentially infinite) data structure representing the input need only be constructed as the program consumes it, which allows interactive programs to be supported.

Nevertheless, it is somewhat less flexible input/output paradigm than the free-wheeling approach offered by imperative languages.

A more fundamental problem is the absence of nondeterminism in functional languages. This gives a problem whenever the time ordering of external events is important, as for example in a multi-user operating system [Hend82], or a program that has to service asynchronous interrupts. It has become clear that some fundamentally new primitive(s) must be introduced to support nondeterministic programming, and a number of suggestions have been made, including **nondeterministic merge** [Park82], **nondeterministic choice** (McCarthy's "amb" operator [McCar63]) [Abram82] [Jones83], **hiatons** [Park82], **timestamps** [Broy83], and **oracles**. No clear consensus has yet emerged on this issue.

## 9.3 Arrays

Most functional languages do not support arrays, which are vital for expressing numerical algorithms. The immediate reason for this is that the updating of arbitrary individual elements of an array is hard to implement efficiently because in principle this operation returns a whole new array (but see Hudak and Bloss [Hudak85] for ways to optimise this).

This problem is more technological than fundamental, and the SISAL development effort has the efficient handling of arrays as a major objective [Univ85].

## 10. Conclusions

We have presented a case for using functional programming languages as a tool to help meet the challenge of writing correct programs.

To this end we reviewed their type systems, means of combination and abstraction, amenability to formal methods, and support for parallel programming.

Language design is only one component of a software engineering methodology, but it has a far-reaching effect (for good or ill) on the rest of the programming environment. We conclude that functional languages merit serious consideration as a key component of advanced program development systems.

\* Unix is a trade mark of Bell Labs.

\* Ada is a trade mark of the US Department of Defence.

## References

- [Abels85] Abelson H and Sussman GJ, "Structure and interpretation of computer programs", MIT Press, 1985.
- [Abram82] Abramsky S, "SECD-M - a virtual machine for applicative multiprogramming", Computer Systems Lab, Queen Mary College, Nov 1982.
- [Augus84] Augustsson L, "A compiler for Lazy ML", Proc ACM Symposium on Lisp and Functional Programming, Austin, Aug 1984.
- [Augus84a] Augustsson L, "A compiler for lazy ML", Proc ACM Lisp Conference, Austin, Aug 1984.
- [Augus85] Augustsson L, "Compiling pattern matching", Functional Programming and Computer Architecture, ed Jounnaud, LNCS 201, Springer Verlag, Aug 1985.
- [Back78] Backus J, "Can programming be liberated from the von Neumann style?", CACM 21(8), pp613-641, Aug 1978.
- [Brook82] Brooks FP, "The mythical man month", Addison Wesley, 1982.
- [Broy83] Broy M, "Applicative real-time programming", Proc 9th IFIP, Information Processing 1983, North Holland, pp259-264, 1983.
- [Burge75] Burge WH, "Recursive programming techniques", Addison Wesley, 1975.
- [Burn85] Burn G, Hankin CL and Abramsky S, "Strictness analysis of higher order functions", Science of Computer Programming (to appear); also DoC 85/6, Dept Comp Sci, Imperial College, London, Apr 1985.
- [Burst77] Burstall, RM, "Design considerations for a functional programming language", Proc Infotech State of the Art Conference, Copenhagen, 1977.
- [Burst77a] Burstall RM and Darlington J, "A transformational system for developing recursive programs", JACM 24(1), pp44-67, Jan 1977.
- [Burst81] Burstall RM, MacQueen DB, Sannella DT, "Hope - an experimental applicative language", Edinburgh report CSR-62-80, 1981.
- [Burst84] Burstall RM and Lampson B, "A kernel language for abstract data types and modules", Semantics of Data Types, Kahn et al, LNCS 173, Springer Verlag, 1984.
- [Burst84a] Burstal RM, "Programming with modules as typed functional programming", Int'l Conf on 5th Generation Computing Systems, Tokyo, Nov 1984.
- [Card85] Cardelli L, "Amber", Bell Labs, Murray Hill, April 1985.
- [Card85a] Cardelli L, "Basic polymorphic typechecking", Polymorphism 2(1), Jan 1985.
- [Card85b] Cardelli L and Wegner P, "On understanding types, data abstraction and polymorphism", Tech Rep CS-85-14, Dept Comp Sci, Brown Univ, Aug 1985.
- [Clack85] Clack CD and Peyton Jones SL, "Generating parallelism from strictness analysis", Internal Note 1679, Dept Comp Sci, University College London, Feb 1985.
- [Clock81] Clocksin and Mellish CS, "Programming in Prolog", Springer Verlag, 1981.

- [Damas82] Damas L and Milner R, "Principal type schemes for functional programs", Proc ACM Symposium on Principles of Programming Languages, pp207-212, 1982.
- [Darl80] Darlington J, "The design of efficient data representations", Imperial College, 1980.
- [Darl81] Darlington J, "The structured description of algorithm derivation", in Algorithmic Languages, ed de Bakker and von Vliet, North Holland, 1981.
- [Darl83] Darlington J, "Unifying logic and functional languages", Imperial College, 1983.
- [Darl84] Darlington J, "Functional programming", in Distributed Computing, ed Chambers et al, Academic Press, 1984.
- [Darl85] Darlington J, Field AJ, Pull H, "The unification of functional and logic programming", Dept of Computer Science, Imperial College, Feb 1985.
- [Dijkstra68] Dijkstra EW, "The GOTO statement considered harmful", CACM 11(3), pp147-148, March 1968.
- [Fages83] Fages F and Huet GP, "Complete sets of unifiers and matches in equational theories", Proc 8th Colloquium on Trees in Algebra and Programming, Springer Verlag LNCS 159, pp205-220, 1983.
- [Fairb82] Fairbairn J, "Ponder and its type system", TR 31, Computer Lab, Cambridge, Nov 1982.
- [Fairb85] Fairbairn J, "A new type checker for a functional language", Science of Computer Programming (to appear), 1985.
- [Fairb85a] Fairbairn J, "Design and implementation of a simple typed language based on the lambda calculus", Tech Rep 75, Computer Lab, Cambridge, May 1985.
- [Glas84] Glaser H, Hankin C and Till D, "Principles of functional programming", Prentice Hall, 1984.
- [Goldf81] Goldfarb W, "The undecidability of the second order unification problem", Theoretical Computer Science 13, pp225-230, 1981.
- [Gord79] Gordon MJC, Milner AJ, Wadsworth CP, "Edinburgh LCF", Springer Verlag LNCS 78, 1979.
- [Hamm84] Hammond K, "The KRC manual", CSA/16/1984, DSAG-3, University of East Anglia, May 1984.
- [Harl84] Harland DM, "User defined types in a polymorphic language", Computer Journal 27(1), pp47-56, Jan 1984.
- [Hend80] Henderson P, "Functional programming", Prentice Hall, 1980.
- [Hend82] Henderson P, "Purely functional operating systems", in Functional Programming and its Applications, ed Darlington, Henderson and Turner, CUP, pp177-192, 1982.
- [Hindl69] Hindley R, "The principal type scheme of an object in combinatory logic", Trans American Mathematical Society 146, pp29-60, 1969.
- [Hudak84] Hudak P and Kranz, "A combinator based compiler for a functional language", Proc 11th Symposium on Principles of Programming Languages, pp122-132, Jan 1984.
- [Hudak85] Hudak P and Bloss A, "The aggregate update problem in functional programming systems", Proc 12th ACM Symposium on Principles of Programming Languages, 1985.

- [Huet73] Huet GP, "The undecidability of unification in third order logic", Information and Control 22, pp257-267, 1973.
- [Huet75] Huet GP, "Unification in the typed lambda calculus", Proc Symposium on the lambda calculus and computer science theory, Springer Verlag LNCS 37, pp192-212, 1975.
- [Hugh82] Hughes RJM, "Graph reduction with supercombinators", PRG-28, Programming Research Group, Oxford, June 1982.
- [Hugh84] Hughes RJM, "Why functional programming matters", PMG-40, Chalmers University of Technology, Goteborg, Sweden, 1984.
- [Hugh85] Hughes RJM, "Strictness detection in non-flat domains", Programming Research Group, Oxford, Aug 1985.
- [Islam81] Islam N, Myers TJ, Broome P, "A simple optimiser for FP-like languages", Proc ACM Conference on Functional Programming Languages and Computer Architecture, New Hampshire, pp33-40, Oct 1981.
- [Johns85] Johnsson T, "Lambda lifting - transforming programs to recursive equations", Functional Programming and Computer Architecture, ed Jounnaud, LNCS 201, Springer Verlag, Aug 1985.
- [Jones83] Jones S, "Abstract machine support for purely functional operating systems", PRG-34, Programming research group, Oxford, Aug 1983.
- [Jones85] Jones ND et al, "An experiment in partial evaluation - the generation of a compiler generator", Proc Conf on Rewriting Techniques and Applications, LNCS, Springer Verlag, 1985.
- [Kieb81] Kieburtz RB, Shultis J, "Transformations of FP program schemes", Proc ACM Conference on Functional Programming Languages and Computer Architecture, New Hampshire, pp41-48, Oct 1981.
- [Kowal79] Kowalski R, "Logic for problem solving", North Holland, 1979.
- [Land66] Landin PJ, "The next 700 programming languages", CACM 9(3), pp157-166, Mar 1966.
- [Linds85] Lindstrom G, "Functional programming and the logical variable", ACM Conference on the Principles of Programming Languages, pp266-280, Jan 1985.
- [MacQ85] MacQueen D, "Modules for Standard ML", Polymorphism 2(2), Oct 1985.
- [McCar63] McCarthy J, "A basis for a mathematical theory of computation", in Computer Programming and Formal Systems, ed Braffort and Hirschberg, North Holland, pp33-70, 1963.
- [Meira85] Meira SRL, "On the efficiency of applicative algorithms", PhD thesis, Computer Lab, University of Kent, March 1985.
- [Miln78] Milner R, "A theory of type polymorphism in programming", Journal of Computer and System Sciences 17(3), pp348-375, Dec 1978.
- [Miln85] Milner R, "The Standard ML core language", Polymorphism 2(2), Oct 1985.
- [Moll84] Moller B (ed), "A survey of the project CIP", TUM-18406, Institut fur Informatik, Technische Universitat Munchen, July 1984.
- [Mycr81] Mycroft A, "Abstract interpretation and optimising transformations for applicative programs", Edinburgh CST-15-81, 1981.
- [Mycr84] Mycroft A and O'Keefe R, "A polymorphic type system for Prolog", Artificial Intelligence 23, Elsevier, pp295-307, 1984.

- [Park82] Park D, "The fairness problem and nondeterminism in computing networks", Proc 4th Advanced Course on Theoretical Computer Science, Mathematisch Centrum, 1982.
- [Parts83] Partsch H and Steinbruggen R, "Program transformation systems", ACM Computing Surveys 15(3), pp199-236, Sept 1983.
- [Peyt83] Peyton Jones SL, "Arbitrary precision arithmetic using continued fractions", Internal Note 1533, University College London, Dec 1983.
- [Reddy85] Reddy US, "On the relationship between logic and functional languages", in Functional and logic programming, ed Degroot and Lindstrom, Prentice Hall, 1985.
- [Robin71] Robinson JA, "The unification computation", in Machine Intelligence 6, Edinburgh University Press, pp63-72, 1971.
- [Robin84] Robinson JA, "Syracuse University Parallel Expression Reduction - first annual report", Syracuse University, Dec 1984.
- [Stoye85] Stoye W, "The implementation of functional languages using custom hardware", PhD thesis, Computer Lab, University of Cambridge, May 1985.
- [Tighe85] Tighe S, "A study of the parallelism inherent in combinator reduction", Parallel processing program, MCC, Austin, Texas, Aug 1985.
- [Turn76] Turner DA, "The Sasl manual", St Andrews, Dec 1976.
- [Turn79] Turner DA, "A new implementation technique for applicative languages", Software Practice and Experience 9, pp31-49, 1979.
- [Turn82] Turner DA, "Recursion equations as a programming language", in Functional Programming and its Applications, ed Darlington, Henderson and Turner, CUP, pp1-28, 1982.
- [Turn83] Turner DA, "Functional programming and proofs of program correctness", in Tools and Notions for Program Construction, Neel (ed), CUP, 1983.
- [Turn85] Turner DA, "The Miranda manual", University of Kent, Canterbury, 1985.
- [Turn85a] Turner DA, "Miranda - a non-strict functional language with polymorphic types", Functional Programming and Computer Architecture, ed Jounnaud, LNCS 201, Springer Verlag, Aug 1985.
- [Univ85] University of Manchester, "SISAL language reference manual", M-146, Computer Science Dept, University of Manchester, Jan 1985.
- [Wadl81] Wadler P, "Applicative style programming, program transformation, and list operators", Proc ACM Conference on Functional Programming Languages and Computer Architecture, New Hampshire, pp25-32, Oct 1981.
- [Wadl85] Wadler P, "Strictness analysis on non-flat domains", Programming Research Group, Oxford, Nov 1985.
- [Wadl86] Wadler P, "Compiling pattern matching and list comprehensions", in Implementation of Functional Programming Languages, Peyton Jones, Prentice Hall, 1986.

# PRACTICAL METHODS AND TOOLS FOR SPECIFICATION

J. Ludewig  
ETH Zürich, Institut für Informatik  
CH 8092 Zürich

## Contents

- 0. Introduction
- 1. Fundamentals
  - 1.1 Life Cycle Model
  - 1.2 Cost Distribution
  - 1.3 Terminology
  - 1.4 Why Semi-formal Specification ?
- 2. Principles of Specifications
  - 2.1 Qualities of Specifications
  - 2.2 Useful Properties of Specifications
  - 2.3 Specification Systems Requirements
  - 2.4 General Structure of a Specification System
- 3. Specification Systems: Some Examples  
(each of the following subheadings consists of three paragraphs:  
.1 The Method; .2 The Language; .3 The Tools)
  - 3.1 **SADT** (Structured Analysis and Design Technique)
  - 3.2 **SA** (Structured Analysis) and **proMod** (Projektmodell)
  - 3.3 **PSL/PSA** (Problem Statement Language / P. S. Analyzer)
  - 3.4 **SREM** (Software Requirements Engineering Methodology)
  - 3.5 **EPOS** (Entwicklungs- und Projektmanagement orientiertes Spezifikationssystem)
  - 3.6 **PRADOS** (Projekt-Abwicklungs- und Dokumentations-System)
- 4. Management Aspects
- 5. Conclusions
- 6. Appendix: References, and Addresses of Suppliers

**Note:** Hans Matheis has used an earlier version of this paper for preparing a paper on languages for real-time software specification. Some of his extensions have been integrated here.

Our descriptions of specification systems are based on the material available to us. This information may not be complete, or up to date. Therefore, we are sorry in case some features are not reported correctly. Please refer to the material available from the suppliers (see 6.2).

## 0. Introduction

This is a course on Specification. Since it is based on experiences in the field of Software Engineering, it applies primarily to Software Specifications. Many observations and reports indicate, however, that, from specification aspects, there is not much difference between information processing systems in general and software in particular. Therefore, most of this course applies also to System Specification.

In the first chapter, some fundamentals are discussed. These include the life cycle model and the distribution of costs over the various activities, some definitions, and a rationale for semi-formal specification. The second chapter provides a general outline of a specification system, whose desirable properties are deduced from the qualities of good specifications. In the third chapter, we present some typical specification systems. The primary goal is to show some typical features of such systems rather than to describe them in detail. The fourth chapter addresses management aspects. In chapter 5, some general conclusions are drawn. The appendix (chapter 6) contains a bibliography on specification, and a list of suppliers.

## 1. Fundamentals

### 1.1 Life Cycle Model

Only very small systems can be built in the same way as primitive peoples build houses. As soon as the system is slightly complex, a systematic approach is necessary. The sequence of steps to be taken from the first idea to operation and further on until the system is discarded, is called the System Life Cycle. Though there are many different life cycle models, they are all based on the distinction between certain activities or phases, namely

analysis and specification  
                   design  
                     implementation  
                       integration  
                         operation and maintenance.

Note that the life cycle may be used as a phase model, or a model of activities, or a list of roles. In the sequel, the second meaning is assumed.

Recently, the life cycle concept has been attacked by several authors, not only because it does not reflect the experiences of many projects, but also because alternative ways of building systems (for instance by prototyping) are ignored. See the references in 6.1.6.

## 1.2 Cost Distribution

About two thirds of the total cost of software are caused by activities which take place when the software is already operational (i.e. during maintenance) (Boehm, 1976). Therefore, every attempt to reduce the high cost of software has to focus on maintenance.

(Note that there is an important difference between maintenance of hardware and of software: while hardware is actually *maintained*, i.e. the original state is conserved or restored, software is corrected, extended, or adapted to new requirements, i.e. it is *modified*. A program is different from its original state after maintenance.)

There are three ways of reducing the need for maintenance:

- reduce need for correction
- reduce effort for modification
- reduce total volume (by using standard components or old software)

A good specification contributes to each of these subgoals. Therefore, the overall goal is not to reduce the effort for specification, but rather to invest more in specification in order to save much more during maintenance (and also during design and implementation).

## 1.3 Terminology

### 1.3.1 Specification

To date, we have not achieved a stable and well recognized terminology in Software Engineering. In the sequel, we use a simple, pragmatic definition of "specification" (from Kramer et al., 1982):

*A description of an object stating its properties of interest. It usually implies that the description should try to be precise, testable, and formal.*

*It is recommended that "specification" be used with some attribute, e.g. requirement specification.*

*Specification is frequently used to mean functional specification which contains both requirements and design aspects. This form of use is imprecise.*

Many more relevant terms are defined by the IEEE (1983).

Specifications are written and read by many people, like analysts, customers, managers, and programmers. Since these people differ greatly in their background, education, and interest, they have usually not the same idea of what a specification should look like. Tools, which can change the representation of a given information automatically, can help to meet the requirements of more than just one single group.

### 1.3.2 The System Triangle

When we talk about programming systems, or specification systems, we distinguish three components, or sets of components, namely methods, languages, and tools.

Methods indicate how to proceed, like recipes in a cookbook. Languages restrict the set of possible statements to a particular universe of discourse, and to a certain syntactical representation. Tools check, store, and transform such statements.

All three are strongly interrelated by the abstract concepts of the (specification-) system. Note that the term "methodology" means "science of methods", though it is often misused for "method". Figure 1 exemplifies the system triangle:

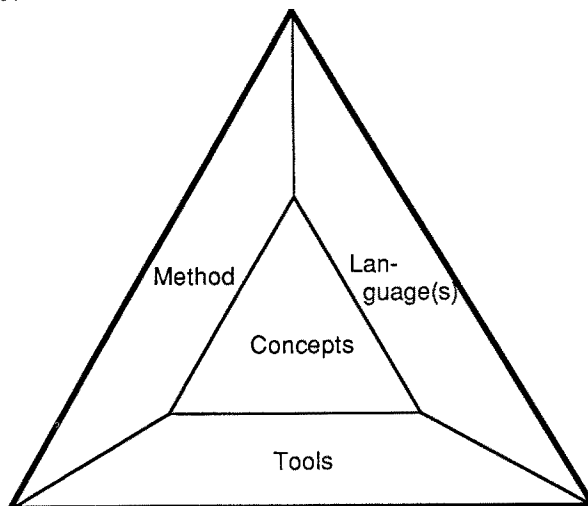


Figure 1: System triangle

### 1.3.3 Levels of Formality

There are languages of various formality. For our purposes, we distinguish four levels of formality, or styles:

| <u>Style</u> | <u>Syntax</u>       | <u>Semantics</u>    | <u>Examples</u>  |
|--------------|---------------------|---------------------|------------------|
| informal     | not (prec.) defined | not (prec.) defined | natur. languages |
| formatted    | restricted          | not (prec.) defined | forms            |
| semi-formal  | defined             | partially defined   | pseudo-code      |
| formal       | defined             | defined             | progr. languages |

For coding programs, we use a formal language. (Though the semantics of most programming languages are not precisely defined, if at all, there is always a translator which provides a de-facto-definition.) All other documents are written in informal language, sometimes on forms. Forms impose certain restrictions to the way natural language is used, and require the user to answer all relevant questions. Semi-formal languages are comparatively new; their first application was in program design languages (pseudo code).

#### 1.4 Why semi-formal Specification ?

This paper does not treat formal specification. This does not mean that formal techniques are not important. However, they are not yet in a state that users in industry could really apply them. Semi-formal specification, i.e. an approach which is based on semi-formal specification languages, has (at least for the time being) several advantages:

- The languages can be learned and understood with limited effort by people who did not have extensive training in formal methods
- Documents resemble those written in natural language
- Incomplete and vague information fits better in such a system

On the other hand, semi-formal specification systems are superior to traditional informal specifications because

- many deficiencies which would be buried in plain text become visible
- it can be stored in, and retrieved from, a data base
- automatic tools can be used for checking and for changing the representation.

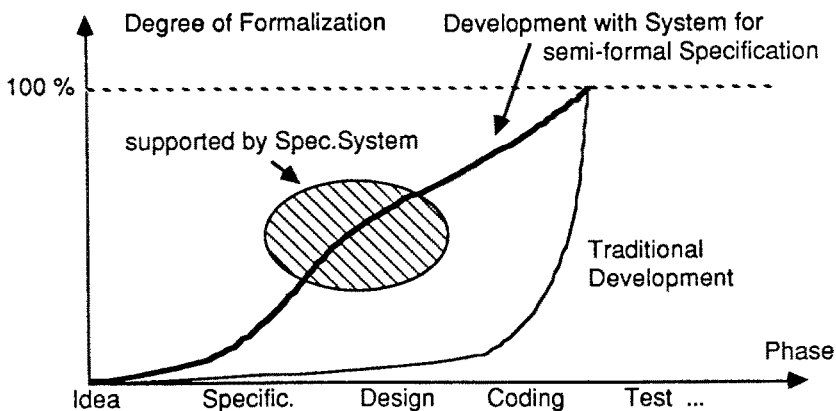


Figure 2: Degree of Formalization during the Software Life Cycle

Figure 2 shows schematically how the software development process is influenced by a system for semi-formal specification. In the traditional approach, there is practically no formalized information until the software is coded. Then, full formalization must be achieved in a single step. This method is, as we all know, error prone, because there are many misunderstandings, inconsistencies, simple errors and other shortages in the specs which are not discovered, because the document produced next, i.e. the code, can only be understood at the level of single instructions. In the modern approach, there are much better chances for detecting deficiencies of the specs, and improving them.

To summarize the message of this paragraph: Specification systems do not shorten the specification phase, but improve the quality of the resulting document.

## **2. Principles of Specification**

### **2.1 Qualities of Specifications**

A specification should be

- correct (i.e. it should reflect the actual requirements)
- complete (i.e. it should comprise all the relevant requirements)
- consistent
- unambiguous
- protected against loss of information and unintended changes
- easily writeable and modifiable
- readable and concise (in order to ease the communication between user and analyst)
- implementable (i.e. it should ease design and implementation)
- verifiable (i.e. there should exist a procedure to check whether or not the product complies with its specs). This quality is also called "testable".
- validateable (i.e. there should be a mechanism to ensure that the specification really reflects the user's specification)
- traceable (i.e. when the specification is changed, it should be easy to identify all statements in other documents affected by that change).

Note that these goals are highly inconsistent. For instance, a formal (e.g. algebraic) specification is verifiable, but not readable for most people, in particular not for the customer. Therefore, it is not validateable.

The first four of the qualities listed above (correctness, completeness, consistency, and unambiguity) do not have the same meaning to all people: vendors of tools for specification, for instance, often claim that their system can guarantee correctness. This does, of course, not imply that the *content* of the specs is correct with respect to the intentions of the customer, but only that certain formal requirements are met. The reason for this is that there is no reference (except the user's brain) to prove specifications correct or complete, in contrast to programs being provably correct with respect to the underlying specification.

## 2.2 Useful Properties of Specifications

In order to achieve the qualities listed above, certain properties are obviously useful:

- The specifications must be recorded on some permanent medium (e.g. paper, magnetic tape).
- They should be as formal as possible, and as informal as necessary. Also, they should support the processing of information which is vague, incomplete, or not yet well defined (i.e. providing a fill-in that indicates the lack of information).
- Specs should exist only in one single copy ("single source concept").
- There should be tools for automatic checks and transformations between different representations.
- Specs must be available in representations appropriate for those who have to use them (e.g. graphical representations which naturally mirror human's way of thinking).

## 2.3 Specification Systems Requirements

From the useful properties stated above, we can derive the requirements of specification systems; such a system should provide

- a data base system as the central information repository,
- a semi-formal specification language and several representations, including a graphical one,
- tools for all clerical tasks (storing, retrieval, checking, transformation).

Since software systems are developed by several people, and usually exist in several versions and variants at the same time, the specification system should also provide

- multi user operation of tools,
- automatic management of versions and variants.

## 2.4 General Structure of a Specification System

As mentioned above, an ideal specification system consists of a method, a language, and a set of tools, which are all based on a common set of concepts. The list following below summarizes the most desirable features.

### Abstract concepts

- Life cycle model
- Stepwise completion
- Permanent validation

### Methods supported by the system

- Enter every information immediately
- Allow for informal texts
- Check early for correctness, completeness, consistency, unambiguity
- Concentrate on information necessary for specification.

### Languages

- Semi-formal specification language
- Several syntactical representations of a specification (e.g. graphics, tables etc.).

### Tools

- Multi-user data base system
- Tools for checking, retrieval and selection.

In reality, however, most systems are incomplete. They are usually based on either of the components, and do never cover the full scope. Some activities started from a particular method (e.g. Structured Analysis, see 3.2), or from a certain representation (e.g. SADT, see 3.1), or from a set of tools (e.g. PRADOS, see 3.6). In the next chapter, some specification systems are presented. Our goal is to give an idea of their dominant feature; we certainly do not attempt to provide complete information. Please refer to the references (6.1), or contact the vendors listed in 6.2.

### 3. Specification Systems: Some Examples

In this chapter, we present some examples of specifications in various languages. Additionally, we briefly describe their underlying methods. The purpose is to show some typical styles rather than to describe systems in detail. These are the examples chosen for this paper:

**SADT** is probably the best known graphical language for expressing specifications;

**Structured Analysis** (SA) is similar to SADT, but has a wider range (towards design). We present it together with **proMod**, a tool which supports SA.

**PSL/PSA** is the classical tool-based specification system.

**SREM** is a very powerful system for describing, and simulating, real time software.

**EPOS**, another tool dedicated to the development of real time systems, is fairly successful in Germany and central Europe.

**PRADOS** was chosen as an example of those systems which do not really form a monolithic specification system, but rather a tool kit.

Our list covers only a part of those systems that we know, which in turn are certainly only a small fraction of those which exist. Therefore, our choice should not be interpreted as a judgement, or recommendation !

#### 3.1 SADT (Structured Analysis and Design Technique)

SADT was developed by SofTech between 1972 and 1975. It covers the requirements analysis, the design and the documentation of specifications, aiming at improved communication between analysts, developers, and users.

##### 3.1.1 The Method

The method SADT focuses on data flow and implies a stepwise refinement of so called SADT-diagrams which are hierarchically ordered. In its original definition (Ross, 1977), there is a duality between so called actigrams and datagrams modelling the data flow in two different ways representing different views of the system:

- actigrams identify functions as central elements of the description and data providing e.g. input or output for the functions
- datagrams identify data as central elements of the description and functions providing e.g. input or output for the data.

The redundancy makes it possible to prove consistency, i.e. one can check whether every function in an actigram is comprised in some datagram, and vice versa.

### 3.1.2 The Language

SADT is a graphical specification language allowing the user to describe the system in terms of activities and data. As outlined above, on the one hand there are actigrams consisting of activities and data. Activities are represented by boxes and data by arrows. On the other hand there are datagrams, where boxes stand for data, while arrows represent activities. Practical experience, however, indicates that most users tend to use only actigrams. For the reason of complexity the language restricts the number of boxes per SADT-diagram to seven.

Figure 3 shows an SADT-box with its typical components:

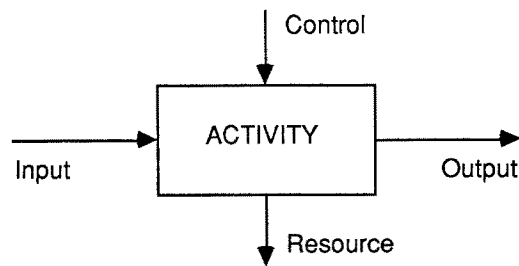


Figure 3: SADT-box (Actigram)

The three actigrams on the following pages (figures 4, 5, 6) show an activity ("ASSIST SADT USERS") at three different levels of refinement. Note that the last actigram (fig. 6) refines an activity ("CREATE KITS") of the second diagram (fig. 5). (Source: Lissandre et al., 1984, from IGL, Paris)

### 3.1.3 Tools

SADT is still a paper and pencil method. And there is no problem in draawing all the diagrams once. However, when there are changes (and the need for change is the only property of software that does never change), diagrams must be redrawn again and again. This is very annoying. Therefore, there have been several activities for providing tool support; the examples in figures 4, 5, 6 were produced by such a tool. Their capabilities range from simple graphics (i.e. they are used as an automatic drawing machine) to fairly sophisticated programs which do some sematic checking and analysis. According to D.T. Ross, who invented SADT, "none (of the tools) is fully successful in implementing SADT" (Ross, 1985 b).

Figure 4: SADT-Diagram, top-level

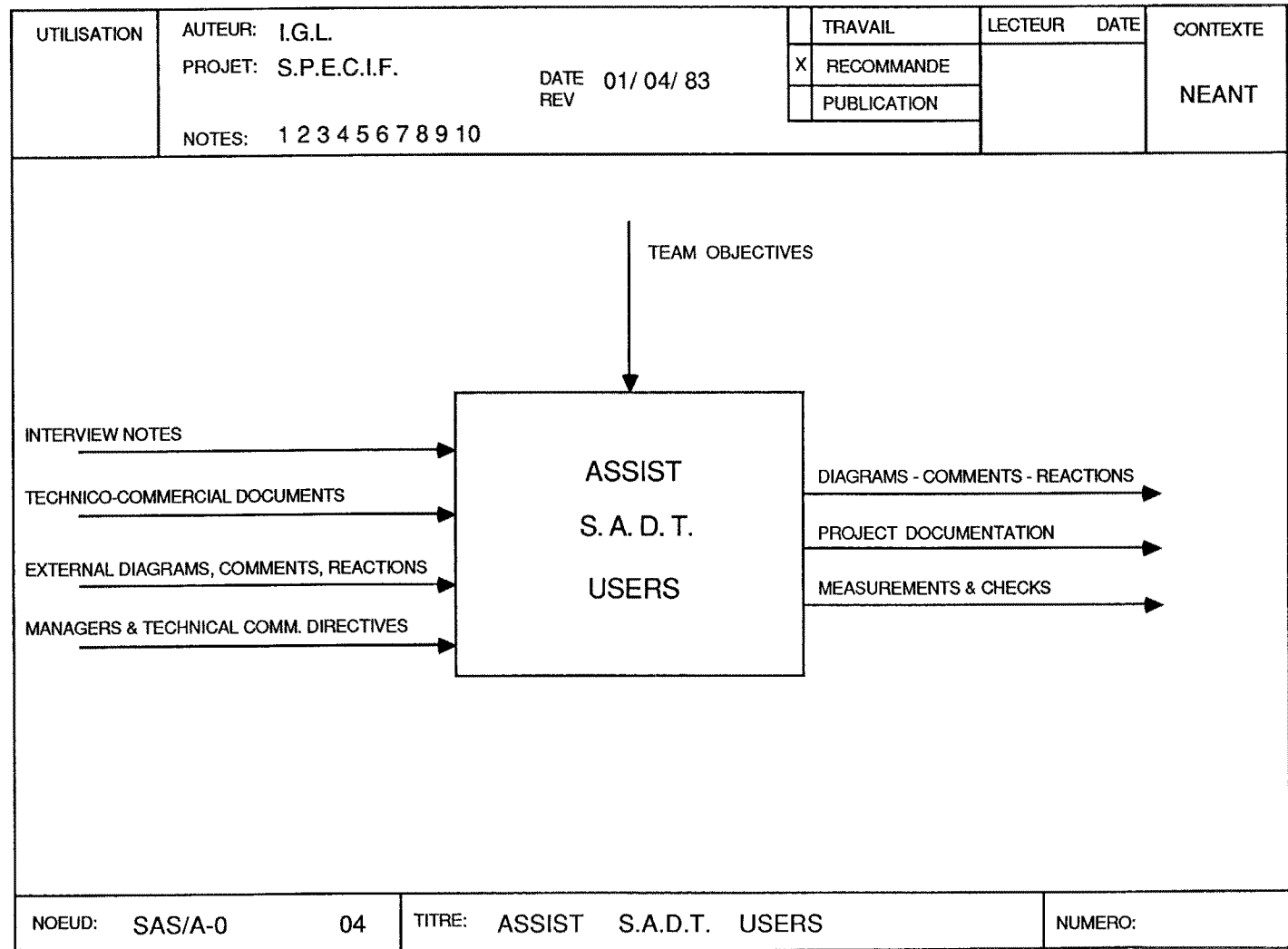
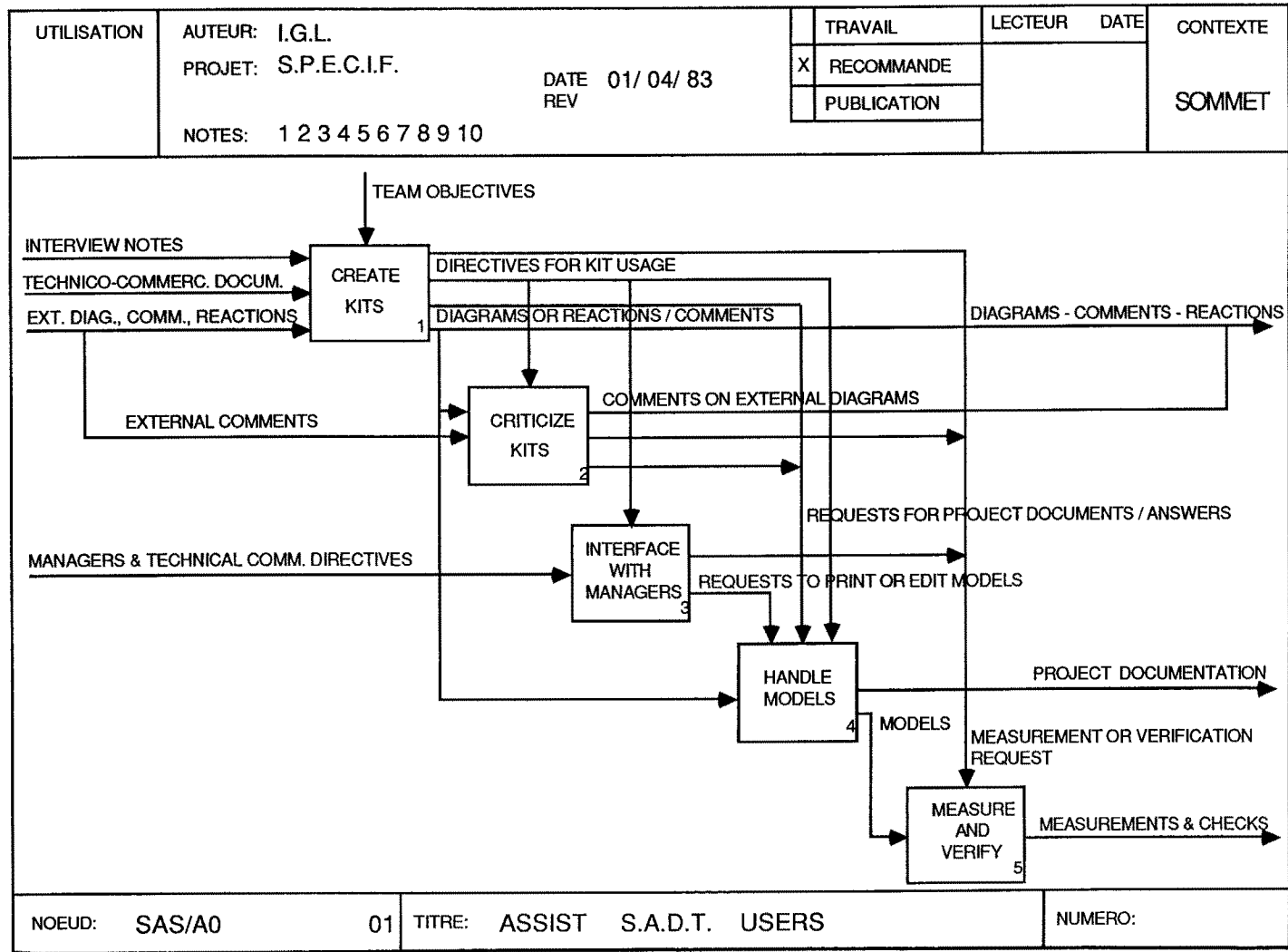


Figure 5: SADT-Diagram, first level below top



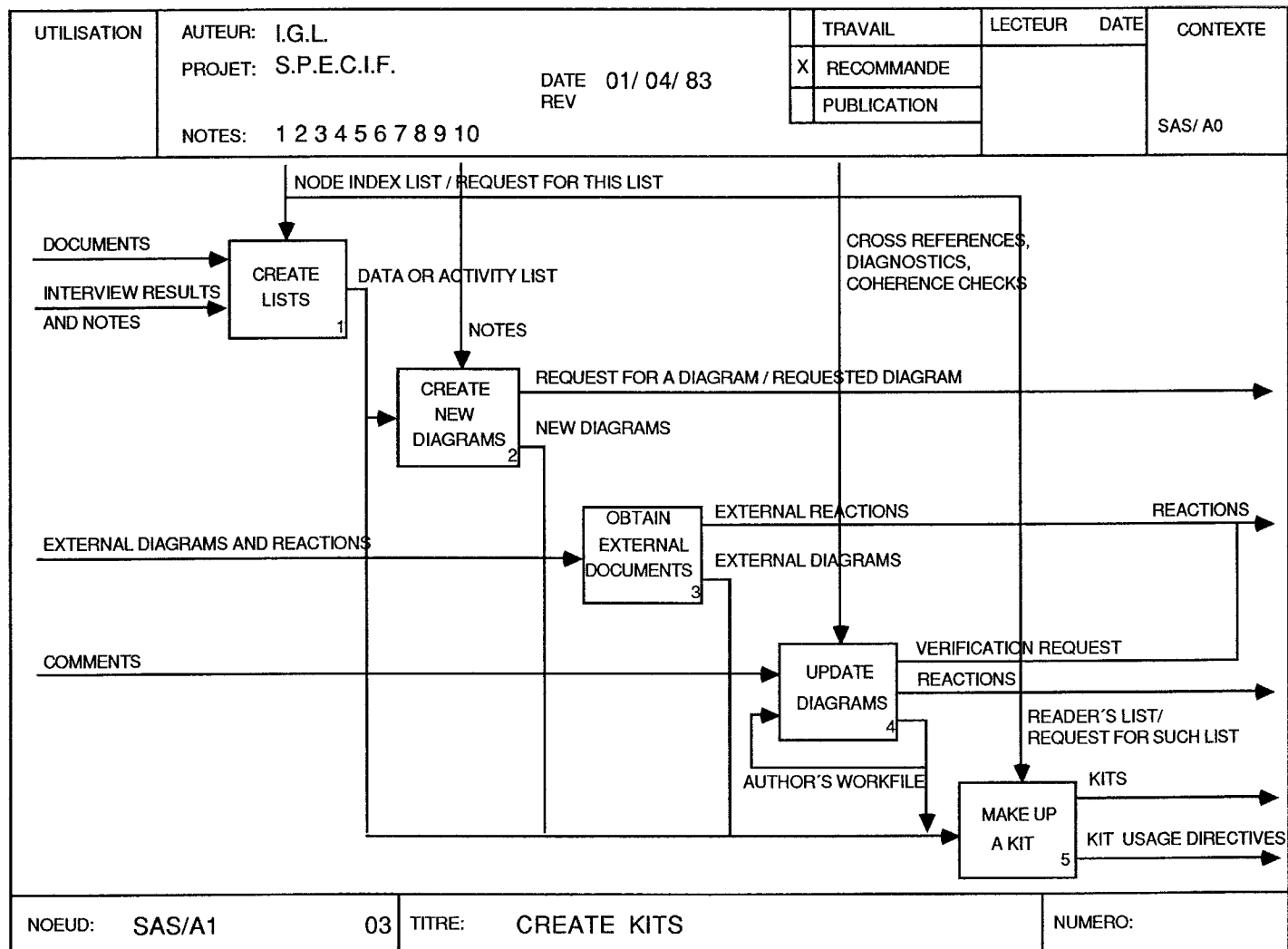
NOEUD: SAS/A0

01

TITRE: ASSIST S.A.D.T. USERS

NUMERO:

Figure 6: SADT-Diagram, second level



## 3.2 Structured Analysis (SA)

SA was developed by Yourdon and others (see Yourdon, Constantine, 1979). Although, the name is very similar to SADT, only the data flow as the central principle is common to both. It is used for analysis and both coarse and detailed design.

### 3.2.1 The Method

The method allows the user to model a system with data-flow diagrams (DFDs) consisting of data, and processes transforming the data. In other words, DFDs describe the flow of data through the system by denoting sources and sinks for data flows, the data flows itself, and processes. So called minispecs are used to describe processes in more detail. For refining the structure of data, a data dictionary (DD) is used. SA proposes a stepwise decomposition of DFDs so that each process in the parent DFD is broken down into several child DFDs. Consequently, several levels of DFDs emerge.

SA proposes two major steps. The first one is to develop a so called context diagram, which shows how the system is connected to its environment. Hereby, the user defines the interface in terms of sources and sinks of the environment, processes, data flows, and files. Note that the data flow consists of both the data and the direction of flow.

In the second step, the user partitions and refines the system "as long as possible", i.e. each process of a DFD is described in more and more detail until the level of atomic processes is reached. Then the user writes minispecs demonstrating the algorithmic structure of these atomic processes. Also, a data dictionary is created containing the structure of the data. SA also gives naming conventions for processes, dataflows, files, which can help the user to express his understanding most clearly.

### 3.2.2 The Language

The sources and sinks belonging to the environment of the target system are shown as boxes on a data-flow diagram. Other symbols are circles representing processes, arrows representing data flows, and bars representing files. Please note that the first time a file is referenced in a DFD two bars are used (see fig. 7a, file "Bit Map") while further references to this file (in other DFDs) are denoted by a single bar (see fig. 8a, file "Bit Map").

The minispecs are written in pseudo-code, the data described in the data dictionary is written in a BNF-like notation.

The examples given below were taken from a paper on the Tektronix-tool (Bell, 1985). They show data-flow diagrams, together with minispecs and information stored in the data dictionary.

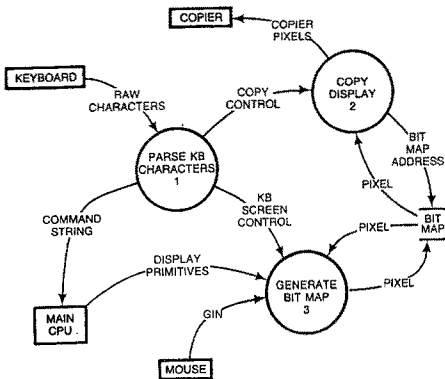


Figure 7a: DFD for a display controller

```
COPIER_PIXEL = PIXEL
BIT_MAP_ADDRESS = INTEGER
PIXEL = LOGICAL
TEXT = ASCII_CHAR
GRAPHICS = [POLYLINE | POLYMARKER | AREA FILL | GDP]
SCREEN_CONTROL = [SCROLL | ERASE | REVERSE | HORIZ_SCROLL]
KB_SCREEN_CONTROL = SCREEN_CONTROL
BIT_MAP = [PIXEL]
BIT_MAP_PIXEL = PIXEL
TEXT_PIXEL = PIXEL
GRAPHICS_PIXEL = PIXEL
COMMAND_STRING = [ASCII_CHAR | DELIMITER]
DISPLAY_PRIMITIVES = GRAPHICS + TEXT + SCREEN_CONTROL
GIN = X_POSITION + Y_POSITION
X_POSITION, Y_POSITION = INTEGER
```

Figure 7b: DD for 7 a

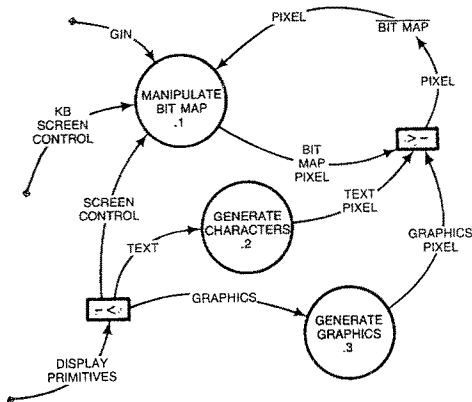


Figure 8a: DFD for Generate Bit Map from figure 7.a

```
CHARACTER_GENERATION_MAP_LOCATION = ASCII_CHAR
FOR I = 1 TO 12 DO
  CHAR_GEN_MAP_INDEX = 1
  FOR J = 1 TO 9 DO
    IF CHARACTER_GEN_MAP_CONTENTS (J) = TRUE
      SEND 1 TO BIT MAP
    ELSE
      SEND 0 TO BIT MAP;
    END
  END
END
```

Figure 8b: Minispec for 8a

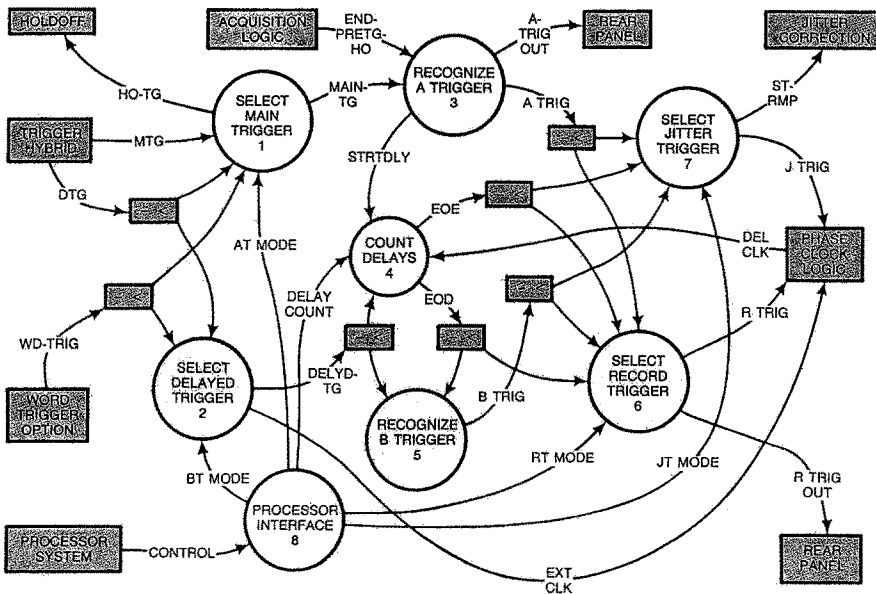


Figure 9: Top level DFD of a trigger gate array

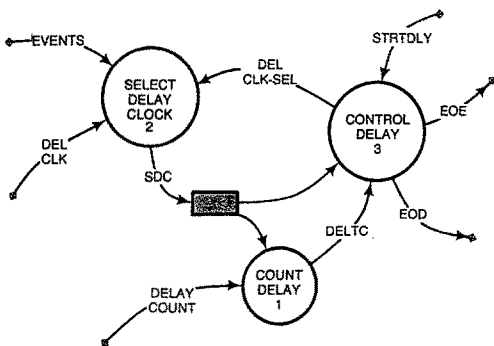


Figure 10: DFD of Count Delays  
(from figure 9)

#### MINISPEC 4.3

**CIRCUIT ELEMENTS:** 2FF2, 2FF3, 2FF4, 2G4, 2G5

**OVERVIEW:** THIS CIRCUIT IS A 3-FLIP-FLOP STATE MACHINE. 2FF2 CONTROLS THE START OF COUNTING DELAY. 2FF3 SETS AT THE END OF EVENTS COUNT, AND 2FF4 SETS AT THE END OF THE TIME-DELAY COUNT. SPECIAL-CASE COUNTS OF NO EVENTS AND 1 EVENT ARE CONTROLLED BY LEVEL INPUTS SET BY THE PROCESSOR. THE INITIAL STATE OCCURS WHEN THE PROCESSOR STROBES RSTACQ. THIS CLEARS 2FF2, WHOSE QBAR OUTPUT CLEARS 2FF4. 2FF3 IS CLEARED BY THE A TRIGGER FLIP-FLOP 1FF1. THE FIRST DCLK AFTER A TRIGGER WILL SET 2FF2 TO ENABLE THE DELAY COUNTER. IF ONEVNT = 1, 2FF3 WILL ALSO SET AT THIS TIME. DCLKS WILL BE COUNTED UNTIL DELTC = 1, CAUSING 2FF3 TO SET. WHEN EOE = 1, THE SELECT DELAY CLOCK LOGIC SWITCHES TO COUNTING DELAY BY TIME. THIS WILL CONTINUE UNTIL THE NEXT OCCURRENCE OF DELTC = 1, WHEN EOD = 1 WILL OCCUR. THE STATE MACHINE REMAINS IN THIS STATE UNTIL THE NEXT RSTACQ.

#### LOGIC:

ALL FLIP-FLOPS ARE RESET ASYNCHRONOUSLY BY PROCESSOR ACTION.

SET STRTDEL = 0 WHEN RSTACQ = 1

SET EOD = 0 WHEN STRTDELB = 1

SET EOE = 0 WHEN ATB = 1

ALL FLIP-FLOPS WILL SET ON CONDITION ON THE RISING EDGE OF DCLK

SET STRTDEL = 1 WHEN AT = 1 (RESETS ARE NOW REMOVED FROM 2FF3, 2FF4)

SET EOE(N+1) = ONEVNT + DELTC + EOE + NOEVNTS

SET EOD(N+1) = (EVDN + EOD) \* (DELTC + EOD) = EVDN \* DELTC + EOD

Figure 11: Minispec of Control Delay  
(from figure 10)

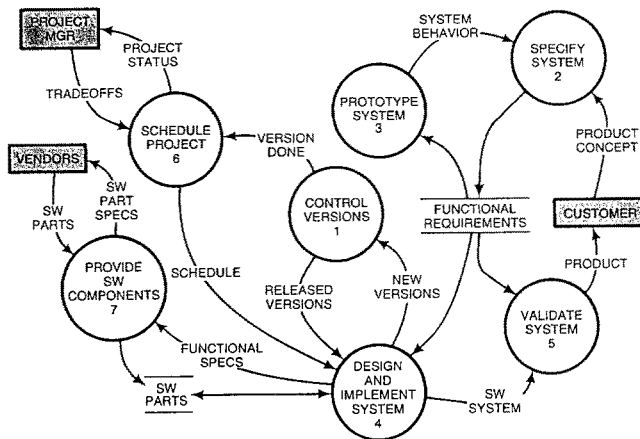


Figure 12: DFD of a product development

### 3.2.3 Tools for SA: proMod (Projektmodell)

proMod was developed bei GEI, Aachen, FRG. It is based on the Structured Analysis/Structured Design-concept.

While the project proceeds, all information is accumulated in the proMod project library.

Tools for Structured Analysis are:

|               |                                                |
|---------------|------------------------------------------------|
| DFD-processor | editing and processing of data flow diagrams   |
| DD-processor  | data dictionary system                         |
| TD-processor  | minispec - processor                           |
| AAD-analyzer  | cross checking between DFDs, DD and minispecs. |

Tools for Structured Design are:

|                |                               |
|----------------|-------------------------------|
| Translator     | from SA - to SD - System      |
| MS - processor | for module specifications     |
| FS - processor | for functional specifications |
| DD - processor | data dictionary system        |
| SE - analyzer  | cross-checking at SD - level  |

Below the level of SD, proMod provides PDL and DARTS, two pseudo-code-systems. Other tools generate code-frames in several languages (PASCAL, FORTRAN, COBOL).

proMod is available on VAX/VMS, and IBM-PC (XT, AT) / PC-DOS.

### 3.3 Problem Statement Language/Problem Statement Analyzer (PSL/PSA)

PSL was developed at the University of Michigan by the ISDOS-project (Information System Design and Optimization System) in the seventies. PSL primarily supports requirements analysis and documentation.

Like some other systems developed at a university, PLS/PSA is now supported, improved, and commercially distributed by a private company (META-systems).

#### 3.3.1 The Method

PSL/PSA was the very first tool-based system for semi-formal specification which was actually useful - and commercially successful. All other such systems are copies of PSL/PSA, at least in part.

PSL/PSA emerged since 1970 in a very organic manner, and Daniel Teichroew and his co-workers did never put too much effort in writing down the method they had in mind. Still, there is a method behind PSL: It is the one sketched in 2.4.

#### 3.3.2 The Language

PSL is based on the entity-relationship approach first described by Chen (Chen, 1976), but applied long before. The entity-relationship model was originally used as a database model splitting the world to be described into entities, and relationships between these entities. The dominant feature of this approach is the similar treatment of entities and relationships.

Different from SADT and SA, PSL is a linear (textual) language. PSL provides some 30 entity-classes and 75 relations to the user. The most important ones are:

##### Entity-classes:

|                   |                                   |
|-------------------|-----------------------------------|
| REAL WORLD ENTITY | objects outside the target system |
| PROCESS           | activities                        |
| INPUT             | input data                        |
| SET               | set of data elements              |

##### Relations:

|           |                                                                             |
|-----------|-----------------------------------------------------------------------------|
| GENERATES | e.g. <process> GENERATES <data>                                             |
| RECEIVES  | e.g. <process> RECEIVES <data>                                              |
| UPDATES   | e.g. <process> UPDATES <data>                                               |
| CONSISTS  | describes data structures;<br>e.g. colour CONSISTS yellow, red, green, blue |

PSA Version A5.2R2M

Jul 23, 1983 20:05:19

PSL/PSA - ISDOS - VM/CMS

## IPSL Input Source Listing

Parameters: DB=VESSEL.DBF INPUT=VESSEL.PSL SOURCE=LISTING NOCROSS-REFERENCE  
 UPDATE DATABASE-REFERENCE NOWARN-NEW-OBJECTS NOSTATEMENT-NUMBERS  
 DBNBUF=200 WIDTH=84 LINES=60 INDENT=0 HEADING PARAMETERS PAGE-CC=ON  
 NOEXPLANATION

## LINE S T M T

```

1 >/* This is a set of PSL statements to define user views */
2 >
3 >/* Here is the global users' view */
4 >
5 >DEF ENTITY      Userviews;
6 >  TKEY          'Global';
7 >  SUBPARTS ARE  User-View-1,
8 >                User-View-2,
9 >                User-View-3,
10 >               User-View-4,
11 >               User-View-5,
12 >               User-View-6,
13 >               User-View-7;
14 >  DESC;
15 >This is a global view of a ship company.;
16 >
17 >
18 >/* ELEMENTs are declared */
19 >
20 >DEF ELE          Vessel,Cargo-Volume,Details,Port,Date-of-Arrival,
21 >                Date-of-Departure,Consignee,Container#,Size,
22 >                Shipping-Agent,Waybill#,
23 >                Delivery-Date,Contents,
24 >                Handling-Instructions;
25 >
26 >
27 >/* Here is the local users' view */
28 >
29 >DEF ENTITY      User-View-1;
30 >  TKEY          'V1';
31 >  CSTS OF       View1-Ship;
32 >  ATTR ARE      FREQUENCY-IS      100,
33 >                TIMING-REQUIREMENT 25;
34 >  RPD IS        'E. Basar';
35 >  DESC;
36 >Information is stored about each ship, including
37 >the volume of its cargo storage capacity.;
38 >
39 >
40 >DEF ENTITY      User-View-2;
41 >  TKEY          'V2';
42 >  CSTS OF       View2-Ship,
43 >               View2-Ship-Port,
44 >               View2-Port;
45 >  ATTR ARE      FREQUENCY-IS      100,
46 >                TIMING-REQUIREMENT 50;
47 >  RPD IS        'E. Basar';
48 >  DESC;

```

Figure 13: PSL source listing (incomplete)

PSA Version A5.2R2M

Jul 23, 1983 20:05:19

PSL/PSA - ISDOS - VM/CMS

## IPSL Input Source Listing

LINE S T M T

```

49 >A ship stops at many ports and it is necessary to
50 >print out its itinerary.;
51 >
52 >
53 >DEF ENTITY      User-View-3;
54 >  TKEY          'V3';
55 >  CSTS OF       View3-Consignee,
56 >                View3-Port,
57 >                View3-Ship,
58 >                View3-Container;
59 >  ATTR ARE      FREQUENCY-IS      25,
60 >                TIMING-REQUIREMENT 7;
61 >  RPD IS        'E. Basar';
62 >  DESC;
63 >Persons who ship goods are referred to as consignees.
64 >Their goods must be crated or stored in shipping containers.
65 >These are given a container identification number. A list
66 >can be obtained, when requested, of what containers have
67 >been sent by a consignee.;
68 >
69 >
70 >DEF ENTITY      User-View-4;
71 >  TKEY          'V4';
72 >  CSTS OF       View4-Agent,
73 >                View4-Port,
74 >                View4-Container;
75 >  ATTR ARE      FREQUENCY-IS      110,
76 >                TIMING-REQUIREMENT 75;
77 >  RPD IS        'Chiang Wan';
78 >  DESC;
79 >The shipments are all handled by shipping agents. A
80 >shipping-agent report must be generated, listing all
81 >the containers that a given agent is handling and giving
82 >their waybill numbers.;
83 >
84 >
85 >DEF ENTITY      User-View-5;
86 >  TKEY          'V5';
87 >  CSTS OF       View5-Waybill,
88 >                View5-Port,
89 >                View5-Ship,
90 >                View5-Container;
91 >  ATTR ARE      FREQUENCY-IS      100,
92 >                TIMING-REQUIREMENT 50;
93 >  DESC;
94 >A waybill related to a shipment of goods between two
95 >ports on a specified vessel. The shipment may consist
96 >of one or more containers.;
97 >
98 >
99 >DEF ENTITY      User-View-6;
100 >  TKEY          'V6';
101 >  CSTS OF       View6-Ship,

```

**Figure 13:** incomplete PSL source listing (continued)



PSA Version A5.2R2M

Jul 23, 1983 20:05:19

PSL/PSA - ISDOS - VM/CMS

## Contents Report

Parameters: DB=VESSEL.DBF FILE=PSANAMES.PSATEMP NOCOMPLETENESS-CHECK  
 NOINDEX NOPUNCHED-NAMES LEVELS=ALL LINE-NUMBERS LEVEL-NUMBERS  
 OBJECT-TYPES PRINT NONEW-PAGE DBNBUF=200 WIDTH=84 LINES=60 INDENT=0  
 HEADING PARAMETERS PAGE-CC=ON NOEXPLANATION

```

1* (ENTITY)      1 User-View-1
1 (GROUP)       2 View1-Ship
2 (ELEMENT)     3 Vessel
3 (ELEMENT)     3 Cargo-Volume
4 (ELEMENT)     3 Details
2* (ENTITY)     1 User-View-2
1 (GROUP)       2 View2-Ship
2 (ELEMENT)     3 Vessel
3 (GROUP)       2 View2-Ship-Port
4 (ELEMENT)     3 Port
5 (ELEMENT)     3 Vessel
6 (ELEMENT)     3 Date-of-Arrival
7 (ELEMENT)     3 Date-of-Departure
8 (GROUP)       3 View2-Ship (M-I)
9 (ELEMENT)     4 Vessel
10 (GROUP)      3 View2-Port (M-I)
11 (ELEMENT)    4 Port
12 (GROUP)      2 View2-Port
13 (ELEMENT)    3 Port
3* (ENTITY)     1 User-View-3
1 (GROUP)       2 View3-Consignee
2 (ELEMENT)     3 Consignee
3 (GROUP)       3 View3-Container (M)
4 (ELEMENT)     4 Container#
5 (ELEMENT)     4 Date-of-Arrival
6 (ELEMENT)     4 Shipping-Agent
7 (GROUP)       4 View3-Port (I)
8 (ELEMENT)     5 Port
9 (GROUP)       4 View3-Ship (M-I)
10 (ELEMENT)    5 Vessel
11 (GROUP)      2 View3-Port
12 (ELEMENT)    3 Port
13 (GROUP)      2 View3-Ship
14 (ELEMENT)    3 Vessel
15 (GROUP)      2 View3-Container
16 (ELEMENT)    3 Container#
17 (ELEMENT)    3 Date-of-Arrival
18 (ELEMENT)    3 Shipping-Agent
19 (GROUP)      3 View3-Port (I)
20 (ELEMENT)    4 Port
21 (GROUP)      3 View3-Ship (M-I)
22 (ELEMENT)    4 Vessel
4* (ENTITY)     1 User-View-4
1 (GROUP)       2 View4-Agent
2 (ELEMENT)     3 Shipping-Agent
3 (GROUP)       3 View4-Container (M)
4 (ELEMENT)     4 Container#
5 (ELEMENT)     4 Waybill#
6 (ELEMENT)     4 Consignee
7 (ELEMENT)     4 Vessel

```

Figure 15: A PSA report (Contents Report)

### 3.4 Software Requirements Engineering Methodology (SREM)

SREM is directly based on PSL/PSA; it was developed by TRW since about 1975. It supports the early phases (analysis, definition, verification, and validation of requirements) of the software development process. It is especially tailored for the development of large, embedded, real-time systems; the U.S. Air Force was the contractor of that project. For more information on SREM, see papers by M. Alford (references in 6.1.4).

#### 3.4.1 The Method

SREM possesses two important features missing from most other methods or languages for specification. Firstly, it allows the stepwise development of specifications beginning with informal descriptions, from which an increasingly formal specification is developed. Secondly, data on performance (estimated or required) of the target system can be formally included in the specification. Since there is a tool for simulating specs, software designers can check early whether or not they will be able to meet response time requirements.

The method (SREM) is applied in seven steps:

1. Define kernel: identify the interface between the system and the environment and describe the data flows and the data-processing units inside the system.
2. Establish baseline: outline the very first description of the system using either the graphical R-Net formalism (R-Net means requirements-net, a stimulus-response network) or the linear language RSL (requirements statement language).
3. Define data: define data input to, and output from, each so called ALPHA (active component); complete, and improve the RSL-specification developed so far; implement Pascal-procedures for ALPHAs.
4. Add project information, and establish traceability: add management informations, e.g. deadlines, milestones, needed tools etc.
5. Simulate functionality: prove syntactical correctness and simulate dynamic behaviour
6. Identify performance requirements: define traceable, testable performance requirements; each path should be constrained by response time and accuracy
7. Demonstrate feasibility: prove that the current design is useful as a basis for a technical realization by means of a analytical feasibility study

### 3.4.2 The Language

SREM offers the user two means of description:

- a graphical language (**R-Nets**) and
- a textual language (**RSL**).

R-Nets are stimulus-response networks describing reactions in a system evoked by events. An R-Net consists of nodes (ALPHAs and SUBNETs) and arcs connecting the nodes. While ALPHAs are functional specifications of processes, SUBNETs are specifications of processes at a lower level of hierarchy. The flow of control is described by some single entry - single exit constructs (AND for parallel execution, OR for a multiway branch, FOR EACH for a loop). Additionally, validation-points can be inserted in order to express performance requirements.

See figure 16 for a list of all symbols used in R-Nets.

RSL is a textual specification language based on the following concepts:

#### Elements

are standard types defining features of each object of such a standard type. For example, MESSAGE, DATA, and FILE are standard types used to describe data; e.g. ALPHAs stand for processes. Elements represent nouns in the language.

#### Relationships

express logical links between Elements, e.g. <data> INPUT TO <alpha>. They represent verbs in the language.

#### Attributes

are used to complete the description of Elements, e.g. <data> INITIAL VALUE <value>. They represent adjectives in the language.

#### Structures

are used to define the sequences of processing steps and represent R-Nets, SUBNETs, and VALIDATION-PATHs in terms of RSL-statements.

|                                 |                                                                                       |
|---------------------------------|---------------------------------------------------------------------------------------|
| ALPHA                           |    |
| AND                             |    |
| ENTRY NODE ON R_NET             |    |
| ENTRY NODE ON SUBNET            |   |
| EVENT                           |  |
| FOR EACH                        |  |
| INPUT_INTERFACE, OUTPUT_INTERF. |  |
| IF OR                           |  |
| CONSIDER OR                     |  |
| SELECT                          |  |
| SUBNET                          |  |
| RETURN                          |  |
| TERMINATE                       |  |
| VALIDATION_POINT                |  |

Figure 16: Symbols in SREM

RSL is also used to enter the R-Nets, which are then automatically drawn.

A few examples are given below. Figure 17 shows a schematic R-Net. In figures 18a and b both the RSL-representation and the flow graph representation of a sample R-Net are exhibited.

(Source: M. W. Alford, Proceedings of the COMPSAC Conferences 1978, 1980).

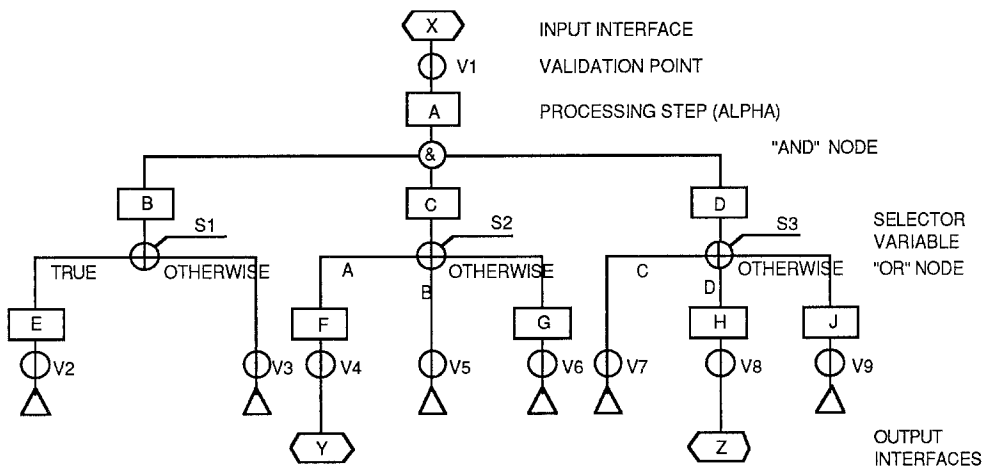


Figure 17: A schematic R-Net

### 3.4.3 The Tool

Like PSL/PSA, SREM is based on a large tool, called REVS (Requirements Engineering Validation System). Beyond the abilities of other tools, REVS allows for project dependent extensions of the specification language, and for simulation of the specs. Maybe that REVS is currently the most powerful tool for specification; but prospective customers in Europe cannot buy it because its distribution is still limited to the U.S.

```

R_NET:  PROCESS_RADAR_RETURN.
STRUCTURE:
  INPUT_INTERFACE RADAR_RETURN_BUFFER
  EXTRACT MEASUREMENT
  DO (STATUS = VALID_RETURN)
    DO UPDATE_STATE AND KALMAN_FILTER END
    DETERMINE_ELEVATION
    DETERMINE_IF_REDUNDANT
    TERMINATE
  OTHERWISE
    DETERMINE_IF_OUTPUT_NEEDED
    DO DETERMINE_IF_REDUNDANT
      DETERMINE_ELEVATION
      TERMINATE
    AND DETERMINE_IF_GHOST
    TERMINATE
  END
END
END.

```

Figure 18a: A sample R-Net, textual representation

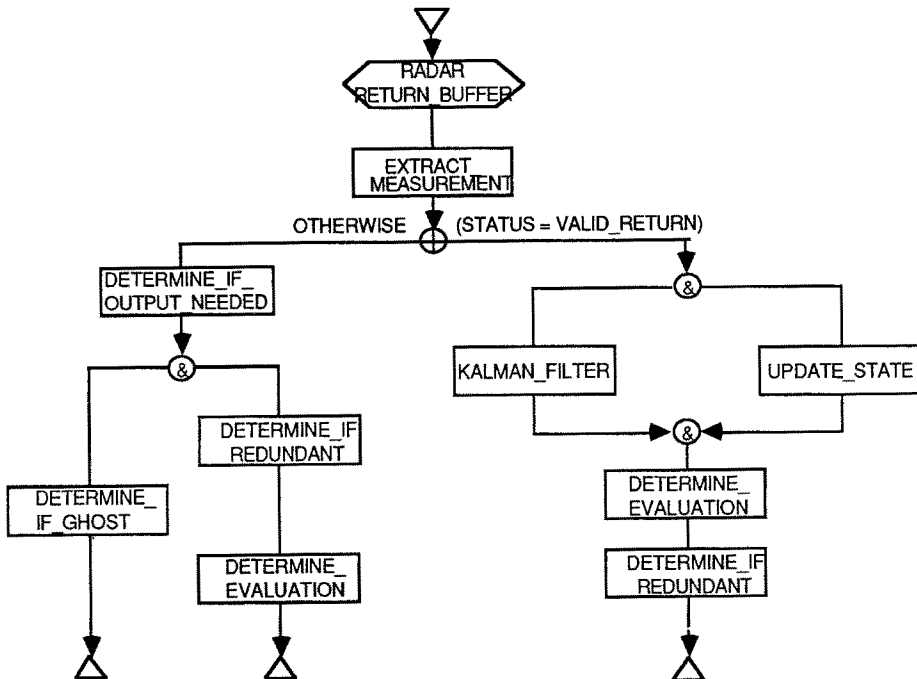


Figure 18b: A sample R-Net, flow graph representation

### 3.5 EPOS

(Entwicklungs- und Projektmanagement orientiertes Spezifikationssystem)

EPOS was developed at TU Stuttgart by R. Lauber and co-workers since 1978 (Biewald et al., 1979). The product is now sold and supported by GPP (see 6.2).

#### 3.5.1 The Method

EPOS is one of the systems which do explicitly not support a particular method (though they do refer to the general principles of SADT).

#### 3.5.2 Languages

In EPOS, there is no clear distinction between languages and tools, i.e. the same name is used both for the language and for the program which is used for processing that language. Therefore, the following list may be inconsistent with other papers on EPOS.

There are three languages used for input:

|        |                                                           |
|--------|-----------------------------------------------------------|
| EPOS-R | language for requirements definition (formatted)          |
| EPOS-S | language for system design (semi-formal)                  |
| EPOS-P | language for project management information (semi-formal) |

Several graphical representations can be generated by the tools.

#### 3.5.3 The Tools

The tools of EPOS are separated in four groups:

|        |                                                                           |
|--------|---------------------------------------------------------------------------|
| EPOS-M | Tools for project management                                              |
| EPOS-A | Analyzer and report generator (for all levels)                            |
| EPOS-D | Generator for documentation (e.g. Petri-Nets, Nassi-Shneiderman-Diagrams) |
| EPOS-C | Human-computer interface of EPOS                                          |

Epos-A stores all information in a (non-standard) data base, which is accessed by all tools.

EPOS is available on PDP 11/RSX 11-M, VAX/VMS, IBM 370/VM/CMS, Intel 8086, 80286/iRMX and several other machines.

### 3.6 PRADOS (Projekt-Abwicklungs- und Dokumentations-System)

PRADOS was (and is still being) developed by SCS, Hamburg and Munich, FRG. It is tailored to UNIX and uses components of the UNIX-System.

#### 3.6.1 The Method

While EPOS is based on a vague idea of a method, which has not been made explicit, PRADOS does not even have such an idea, because it is a collection of tools many of which existed before PRADOS was developed.

#### 3.6.2 The Languages

The languages of PRADOS are the languages of its tools. When SADT is available, there will be at least one genuine specification language in the system.

#### 3.6.3 The Tools

Three general components are:

|           |                          |
|-----------|--------------------------|
| UNIFY     | relational database      |
| XED       | text processing system   |
| IFE-GRAPH | business graphics system |

Upon these, all other components are built:

|         |                                                     |
|---------|-----------------------------------------------------|
| PV, PM  | project management tool                             |
| PB      | project library                                     |
| TE      | text processing                                     |
| MB      | methods library                                     |
| SADT    | SADT Generator (announced)                          |
| DSA     | data description based on entity-relationship-model |
| ESS     | design specification language processor             |
| PS      | pseudo code pre-compiler                            |
| ST      | Nassi-Shneiderman-chart-generator                   |
| BT & TR | test monitor                                        |

## **4. Management Aspects**

There are two important management aspects in the topic of this paper:

First, the decision to use a specification system, and the choice of a particular product, require a commitment of the management. Introduction of a specification system is very expensive. The cost of the system itself and,

possibly, of new hardware is often high, but it is usually negligible compared to the cost of training (or the failures due to insufficient training). The step to using a specification system is of similar importance like the step to using a computer; if you are not prepared to do it right, don't do it at all! Problems are inevitable, and there will be a situation when an important project seems to be late, because it is done with a specification system. If the management is not prepared to show a bold front against the breakers, they will not succeed.

Second, the specification system may improve quality assurance and project control. Most vendors advertise some management tools as part of their products. To date, these are not very powerful. The real improvement stems from the disciplin and standardization implied by the application of a specification system. This side effect is in fact the main advantage of a specification system!

## 5. Conclusions

- There are many specification systems commercially available. Everybody who uses any of the more common machines, and operating systems, will find a specification system, if he or she wants to.
- It is obviously still possible to produce software (and systems) without a specification system. Special problems, like developing user interfaces, are actually better done by other approaches, e.g. prototyping.
- A specification system causes large expenses, mainly for training, but can improve quality and productivity significantly. Therefore, it should be regarded as a (medium- or long-range) investment.
- A specification system improves standardization in the way that every member of a project uses the same method, the same language, and the same tool. Moreover, the documents itself have standardized features. This implies a discipline which is the real benefit of a specification system!
- Vendors say little about the methods, which are most important for the customers.
- Maintenance of specifications is not yet supported by the tools. Therefore, the responsibility to change all documents, when one is modified, rests with the user. If he or she fails to do so (what is the normal situation), the specification becomes obsolete.
- Implementing one's own specification system is hardly feasible, because it takes at least ten person years to develop just a prototype.

## 6. References and Addresses of Suppliers

### 6.1 References

#### 6.1.1 Textbooks on Software Engineering

Boehm, B. W. (1980): **Software Engineering Economics**.  
Prentice Hall, Englewood Cliffs, N.J.

Fairley, R. (1985): **Software Engineering Concepts**.  
McGraw-Hill Book Company, New York usw.

Sommerville, I. (1985): **Software Engineering**. Addison-Wesley Publishing  
Company, London etc., 2nd ed.

#### 6.1.2 The Life Cycle

Lehman, M.M. (1980): Programs, life cycles, and laws of software evolution.  
**Proc. of the IEEE**, **68**, 9, 1060-1076.

Ludewig, J. (1982): Computer aided specification of process control software.  
**IEEE COMPUTER**, **15**, 5, 12-20.

Swartout, W., R. Balzer (1982): On the inevitable intertwining of  
specification and implementation. **Commun. ACM**, **25**, 7, 438-440.

#### 6.1.3 Fundamentals and Principles of Specification

Balzer, R., N. Goldman (1979): Principles of good software specification and  
their implications for specification languages.  
in **Proceedings of Specification of Reliable Software (SRS)**,  
IEEE Cat. No. 79 CH 1402-9C, pp.58-67.

Boehm, B.W. (1976): Software Engineering.  
**IEEE Transactions on Computers**, **C-25**, pp.1226-1241.

Brooks, W.D. (1981): Software Technology Payoff: Some statistical evidence.  
**Journal of Systems and Software**, **2**, 3-9.

IEEE (1983): Standard glossary of software engineering terminology.  
**IEEE Std 729-1983**.

Kramer, J. (ed.) (1982): Glossary of terms.  
**TC on Application Oriented Specification**.  
Jeffrey Kramer, Imperial College, 180 Queen's Gate, GB - London SW7 2BZ.

Parnas, D.L. (1977): The use of precise specifications in the development of software. in Gilchrist, B. (ed.): **Information Processing 77**. North Holland Publishing Company, Amsterdam, New York, Oxford, pp.861-867.

Timm, M. (1982): **Grundlagen von Anforderungs- und Entwurfsspezifikationen im Prozeß der Software-Entwicklung**. GMD-Studien, Nr. 66, 82 S.

#### 6.1.4 Surveys (Articles and Books)

Balzert, H. (1982): **Die Entwicklung von Software-Systemen**. Reihe Informatik/34, Bibliographisches Institut, Mannheim.

Hommel, G. (Hrsg.) (1980): **Vergleich verschiedener Spezifikationsverfahren, am Beispiel einer Paketverteilanlage**. KfK-PDV 186, Teile 1 und 2, Kernforschungszentrum Karlsruhe, BRD.

Cheng, L.L. (1978): Program design languages - an introduction. Report No. ESD-TR-77-324, Electronic Systems Division, Hanscom Air Force Base, MA 01731.

COMPUTER (1982): Special issue on application oriented specification. **IEEE COMPUTER** 15, 5 (May 1982), 10-59.

COMPUTER (1985): Special issue on requirements engineering environments. **IEEE COMPUTER** 18, 4 (April 1985), 9-91.

IEEE-SE (1977): Special collection on requirements analysis. **IEEE Trans. Software Eng.**, SE-3, 2-84.

Ludewig, J., W. Streng (1978):  
Methods and tools for software specification and design - a survey.  
**EWICS TC on Safety and Security**, Paper No. 149, 23 Seiten.

Ludewig, J. (Hrsg.) (1983): **Spezifikation von Realzeit-Systemen - Konzepte, Lösungen, Erfahrungen**. 54. Tagung der Schweizerischen Gesellschaft für Automatik (SGA-ASSPA), Baden/Aargau, 1983- 3-21.

Ohno, Y. (ed.) (1982): **Requirements Engineering Environments**. Proceedings of the International Symposium on current issues of Requirements Engineering Environments; Kyoto, Japan, September 20-21, 1982. NHPC, Amsterdam usw.  
(some of the papers have also been published in COMPUTER, 1985).

Prentice, D. (1981): An analysis of software development environments.  
**ACM SIGSOFT Software Engineering Notes**, 6, No.5, 19-27.

Ramamoorthy, C.V., H.H. So (1977): Survey of principles and techniques of software requirements and specifications.  
 in **Software Engineering Techniques**, Vol.2, Infotech Intern. Ltd.,  
 Nicholson House, Maidenhead, Berkshire, England, pp.265-318.

#### 6.1.5 Particular Specification Methods and Systems

Alford, M. (1977): A requirements engineering methodology for real time processing requirements.  
**IEEE Transactions on Software Engineering**, SE-3, 60-69. (on SREM)

Alford, M. (1985): SREM at the age of eight: The distributed computing design system. **IEEE COMPUTER** 18, 4, 36-46.

Balzert, H. (1985):  
**Moderne Software-Entwicklungssysteme und Werkzeuge.**  
 Reihe Informatik/44, Bibliographisches Institut, Mannheim.  
 (contains material on proMod, PRADOS and other systems; in German)

Bell, R. (1985): Structured analysis aids in micro-computer system design.  
**EDN**, March 21, 1985, 251-257. (on Structured Analysis)

Biewald, J., P. Göhner, R. Lauber, H. Schelling (1979): EPOS - a specification and design technique for computer controlled real-time automation systems. **4th Intern. Conf. on Software Engineering**, München, 1979, IEEE Cat. No. 79 CH 1479 - 9C, pp.245-250.

Hamilton, M., S. Zeldin (1976): Higher Order Software - a methodology for defining software.  
**IEEE Transactions on Software Engineering**, SE-2, 9-32. (on HOS)

Lissandre, M., P. Lagier, A. Skalli, H. Massié (1984): SPECIF - A specification assistance system. Institut de Génie Logiciel, Paris, France. (SADT-tool)

Ludewig, J., M. Glinz, H.J. Huser, G. Matheis, H. Matheis, M.F. Schmidt (1985): SPADES - A Specification and Design System and its Graphical Interface.  
**8th Intern. Conf. on Software Engineering**, IEEE  
 CH2139-4/85/0000/0083, 83-89.

Ross, D.T. (1977): Structured analysis (SA): A language for communicating ideas. **IEEE Trans. on Software Engineering**, SE-3, 16-34. (on SADT)

Ross, D.T. (1985 a): Applications and extensions of SADT.  
**IEEE COMPUTER 18**, 4, 25-34.

Ross, D.T. (1985 b): Douglas Ross talks about Structured Analysis.  
**IEEE COMPUTER 18**, 7, 80-88. (on SADT)

Teichroew, D., E.A. Hershey III (1977): PSL/PSA: a computer aided technique for structured documentation and analysis of information processing systems. **IEEE Trans. Software Eng., SE-3**, 41-48.

Yourdon, E., L.L. Constantine (1979): **Structured Design: Fundamentals of a disciplin of computer programs and systems design**.  
 Prentice Hall Inc., Englewood Cliffs.

#### 6.1.6 Use of Programming Languages for Specification, Prototyping

Goldsack, S.J. (ed.) (1985): **Ada for specification: Possibilities and limitations**. Cambridge University Press (for the Commission of the EC).

Boehm, B.W., T.E. Gray, Th. Seewald (1984): Prototyping versus Specifying: A multi-project experiment. **7th ICSE**, Orlando, FL., March 1984, 473-484;  
 also in **IEEE Trans. on SE, SE-10**, 290-303.

Budde, R. K. Kuhlenkamp, L. Mathiassen, H. Züllighoven (eds.) (1984):  
**Approaches to Prototyping**. Springer-Verlag, Berlin etc.

#### 6.1.7 Software Engineering Environments

Howden, W. (1982): Contemporary software development environments.  
**Comm. ACM**, 25, 5, 318-329.

Hünke, H. (ed.) (1981): **Software Engineering environments**.  
 Proc. of the Symposium held at Lahnstein, June 16 - 20, 1980.  
 North Holland Publishing Company, Amsterdam, New York, Oxford.

Osterweil, L. (1981): Software environment research: directions for the next five years. **IEEE COMPUTER**, April 1981, 35-43.

#### 6.2 Adresses of Vendors

Please note that the following list is rather arbitrary, and far from complete, and it does not imply a judgement or recommendation !

**EPOS** (Entwurfsunterstützendes Prozeß-Orientiertes Spezifikationssystem)  
GPP, Kolpingring 18a, D 8024 Oberhaching. Tel. D 089 - 61 10 42 18

**HOS** (Higher Order Software) and **USE.IT**  
Higher Order Software, Inc., 2067 Massachusetts Avenue  
Cambridge, Massachusetts 02140, USA, Tel. USA 617-661-8900

**MASCOT** (Modular Approach to Software Construction, Operation, and Test)  
MASCOT Suppliers Association  
c/o Computing Standards Section, Room L303, Royal Signals and Radar  
Establishment, St. Andrews Rd., Malvern, Worcestershire, WR14 3PS, GB

**Perspective**  
Software Technology Centre, System Designers Ltd.,  
Systems House, 1 Pembroke Broadway,  
Camberley, Surrey GU15 3XH Great Britain, Tel. GB (0276) 62244

**PET** (Programm-Entwicklungs-Terminalsystem)  
PHILIPS AG, Data Syst., Allmendstr. 140, 8027 Zürich, Tel. CH 01 432211

**PRADOS** (Projekt-Abwicklungs- und Dokumentationssystem)  
SCS Techn. Autom. und Systeme GmbH,  
Hörselbergstr.3, D 8000 München 80, Tel. D 089 - 41 27 - 0

**proMod** (Projektmodell)  
GEI, Albert Einstein Str. 61, D-5100 Aachen, Tel. D 02408 - 130

**PSL/PSA** (Problem Statement Language/Analyzer) and related systems  
META-systems, 315 E. Eisenhower Pkwy., Suite 200,  
Ann Arbor, MI 48104, USA; Tel. USA (313) 663-6027

**Softool** (Softw. Management, Developm., Maintenance, and Conversion Tools)  
via mbp, Semerteichstr. 47, D 4600 Dortmund 1, Tel. D 0049 231 43480  
or directly from SOFTOOL Co., 340 S. Kellogg Av., Goleta, CA 93117

**SPECIF** (Specification System) for SADT-Diagrams  
Institut de Génie Logiciel (IGL), 39 rue de la Chaussée d'Antin,  
F-75009 Paris, France; Tel. F (33) 1 281 41 33

**Tektronix SA Tools** (Structured Analysis)  
Tektronix Inc, P.O.B. 500, Beaverton, OR 97077, Tel. USA (503) 627-7111

**TOPAS-N** (alias **NET**, an editor and simulator for extended Petri-Nets)  
**TOPAS-B** (formerly **BOIE**, a tree-oriented development tool)  
PSI GmbH, Heilbronner Straße 10, D 1000 Berlin 31, Tel. 0049 30 890090